

VMS

---

digital

VMS RTL Library (LIB\$) Manual

Order Number AA-LA76A-TE

# VMS RTL Library (LIB\$) Manual

Order Number: AA-LA76A-TE

**April 1988**

This manual documents the library routines contained in the LIB\$ facility of the VMS Run-Time Library.

**Revision/Update Information:** This document supersedes the LIB\$ section of the *VAX/VMS Run-Time Library Routines Reference Manual*, Version 4.4.

**Software Version:** VMS Version 5.0

**digital equipment corporation  
maynard, massachusetts**

---

**April 1988**

---

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

---


Copyright ©1988 by Digital Equipment Corporation

All Rights Reserved.  
Printed in U.S.A.

---

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	

ZK4609

---

**HOW TO ORDER ADDITIONAL DOCUMENTATION  
DIRECT MAIL ORDERS**

**USA & PUERTO RICO\***

Digital Equipment Corporation  
P.O. Box CS2008  
Nashua, New Hampshire  
03061

**CANADA**

Digital Equipment  
of Canada Ltd.  
100 Herzberg Road  
Kanata, Ontario K2K 2A6  
Attn: Direct Order Desk

**INTERNATIONAL**

Digital Equipment Corporation  
PSG Business Manager  
c/o Digital's local subsidiary  
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

\* Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminister, Massachusetts 01473.

---

---

## Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript<sup>®</sup> printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

---

<sup>®</sup> PostScript is a trademark of Adobe Systems, Inc.



---

# Contents

---

PREFACE	xvii
NEW AND CHANGED FEATURES	xxi

---

CHAPTER 1 OVERVIEW OF THE LIB\$ FACILITY	1-1
--	-----

---

CHAPTER 2 ACCESS TO VMS SYSTEM COMPONENTS	2-1
---	-----

---

2.1 SYSTEM SERVICE ACCESS ROUTINES	2-1
2.2 ACCESS TO THE COMMAND LANGUAGE INTERPRETER	2-2
2.2.1 Obtaining the Command Line	2-3
2.2.2 Chaining from One Program to Another	2-5
2.2.3 Executing a CLI Command	2-6
2.2.4 Using Symbols and Logical Names	2-8
2.2.5 Disabling and Enabling Control Characters	2-8
2.2.6 Creating and Connecting to a Subprocess	2-9
2.3 ACCESS TO VAX MACHINE INSTRUCTIONS	2-9
2.3.1 Variable-Length Bit Field Instruction Routines	2-10
2.3.2 Integer and Floating-Point Routines	2-12
2.3.3 Queue Access Routines	2-12
2.3.4 Character String Routines	2-14
2.3.5 Miscellaneous Instruction Routines	2-16
2.4 PROCESSWIDE RESOURCE ALLOCATION ROUTINES	2-16
2.4.1 Allocating Logical Unit Numbers	2-17
2.4.2 Allocating Event Flag Numbers	2-17
2.5 PERFORMANCE MEASUREMENT ROUTINES	2-18
2.6 OUTPUT FORMATTING CONTROL ROUTINES	2-20
2.7 MISCELLANEOUS INTERFACE ROUTINES	2-22
2.7.1 Indicating Asynchronous System Trap in Progress	2-22

## Contents

2.7.2	Assigning an I/O Channel Along with a Mailbox _____	2-23
2.7.3	Create a Directory or Subdirectory _____	2-24
2.7.4	File Searching Routines _____	2-24
2.7.5	Insert Entry in a Balanced Binary Tree _____	2-31
2.7.6	Common I/O Routines _____	2-35

---

## CHAPTER 3 DATE/TIME MANIPULATION 3-1

---

3.1	DATE/TIME UTILITY ROUTINES	3-2
3.2	DATE/TIME MANIPULATION ROUTINES	3-2
3.3	DATE/TIME FORMATTING ROUTINES	3-3
3.3.1	Date/Time Logical Initialization _____	3-3
3.3.2	Selecting a Format _____	3-4
3.3.2.1	Run-Time Format Mnemonics • 3-5	
3.3.2.2	Specifying Formats at Run Time • 3-6	
	3.3.2.2.1 Specifying Input Formats at Run Time • 3-7	
	3.3.2.2.2 Specifying Output Formats at Run Time • 3-9	
3.3.2.3	Specifying Formats at Compile Time • 3-11	
	3.3.2.3.1 Specifying Input Format Mnemonics at Compile Time • 3-12	
	3.3.2.3.2 Specifying Output Formats at Compile Time • 3-13	
3.3.3	The LIB\$CONVERT_DATE_STRING Routine _____	3-13
3.3.4	The LIB\$GET_DATE_FORMAT Routine _____	3-14
3.3.5	User-Defined Output Formats _____	3-14

---

## CHAPTER 4 CONDITION HANDLING ROUTINES 4-1

---

4.1	AN OVERVIEW OF THE VAX CONDITION HANDLING FACILITY	4-1
4.1.1	Exception Conditions _____	4-4
4.1.2	The Condition Value _____	4-5
4.1.3	Signaling _____	4-7
4.1.3.1	Signal Argument Vector • 4-9	
4.1.3.2	Mechanism Argument Vector • 4-11	
4.1.4	Condition Handlers _____	4-13
4.1.4.1	Default Condition Handlers • 4-13	
4.1.4.2	Possible Condition Handler Actions • 4-14	
4.1.4.3	Interaction Between Default and User-Supplied Handlers • 4-15	
4.1.5	Displaying Messages _____	4-16
4.1.6	Multiple Active Signals _____	4-18

<b>4.2</b>	<b>USING THE VAX CONDITION HANDLING FACILITY</b>	<b>4-20</b>
4.2.1	Establishing a Condition Handler _____	4-20
4.2.2	Writing a Condition Handler _____	4-20
4.2.2.1	Continuing Execution • 4-21	
4.2.2.2	Resignaling • 4-22	
4.2.2.3	Unwinding the Call Stack • 4-22	
4.2.3	Generating Signals _____	4-24
4.2.4	Signaling User-Defined Messages _____	4-26
4.2.5	Logging Error Messages to a File _____	4-27
<hr/>		
<b>4.3</b>	<b>RUN-TIME LIBRARY CONDITION HANDLING ROUTINES</b>	<b>4-29</b>
4.3.1	Convert a Floating-Point Fault to a Floating-Point Trap _____	4-29
4.3.2	Change a Signal to a Return Status _____	4-29
4.3.3	Change a Signal to a Stop _____	4-30
4.3.4	Match Condition Values _____	4-30
4.3.5	Correct a Reserved Operand Condition _____	4-30
4.3.6	Decode the Instruction That Generated a Fault _____	4-30
<hr/>		
<b>4.4</b>	<b>HOW RUN-TIME LIBRARY ROUTINES HANDLE EXCEPTIONS</b>	<b>4-30</b>
4.4.1	Exception Conditions Signaled from Mathematics Routines .	4-31
4.4.1.1	Integer Overflow and Floating-Point Overflow • 4-31	
4.4.1.2	Floating-Point Underflow • 4-31	
4.4.2	Overflow/Underflow Detection Enabling Routines _____	4-32
<hr/>		
<b>CHAPTER 5</b>	<b>MEMORY ALLOCATION ROUTINES</b>	<b>5-1</b>
<hr/>		
<b>5.1</b>	<b>OVERVIEW</b>	<b>5-1</b>
5.1.1	Virtual Address Space _____	5-1
5.1.2	Memory Allocation Routines _____	5-2
<hr/>		
<b>5.2</b>	<b>ALLOCATING AND FREEING PAGES</b>	<b>5-4</b>
<hr/>		
<b>5.3</b>	<b>ZONES</b>	<b>5-6</b>
5.3.1	Zone Attributes _____	5-8
5.3.2	The Default Zone _____	5-12
5.3.3	Zone Identification _____	5-12
5.3.4	Creating a Zone _____	5-13
5.3.5	Deleting a Zone _____	5-13
5.3.6	Resetting a Zone _____	5-14



## Contents

5.4	ALLOCATING AND FREEING BLOCKS	5-14
5.5	ALLOCATION ALGORITHMS	5-15
5.5.1	The First Fit Algorithm	5-16
5.5.2	The Quick Fit Algorithm	5-16
5.5.3	The Frequent Sizes Algorithm	5-16
5.5.4	The Fixed Size Algorithm	5-16
5.6	USER-DEFINED ZONES	5-16
5.7	INTERACTIONS WITH OTHER RUN-TIME LIBRARY ROUTINES	5-19
5.8	INTERACTIONS WITH VMS SYSTEM SERVICES	5-20
<b>CHAPTER 6 DEBUGGING PROGRAMS THAT USE VIRTUAL MEMORY ZONES</b>		<b>6-1</b>
<b>CHAPTER 7 IMAGE INITIALIZATION AND TERMINATION</b>		<b>7-1</b>
7.1	IMAGE INITIALIZATION	7-1
7.2	INITIALIZATION ARGUMENT LIST	7-3
7.3	DECLARING INITIALIZATION ROUTINES	7-4
7.4	DISPATCHING TO INITIALIZATION ROUTINES	7-5
7.5	INITIALIZATION ROUTINE OPTIONS	7-5
7.6	AN EXAMPLE	7-5
7.7	IMAGE TERMINATION	7-6

<b>CHAPTER 8 CROSS-REFERENCE ROUTINES</b>		<b>8-1</b>
<b>8.1</b>	<b>USING THE CROSS-REFERENCE ROUTINES</b>	<b>8-1</b>
<b>8.2</b>	<b>\$CRFCTLTABLE MACRO</b>	<b>8-2</b>
<b>8.3</b>	<b>\$CRFFIELD MACRO</b>	<b>8-3</b>
<b>8.4</b>	<b>\$CRFFIELDEND MACRO</b>	<b>8-4</b>
<b>8.5</b>	<b>CROSS-REFERENCE OUTPUT</b>	<b>8-5</b>
<b>8.6</b>	<b>EXAMPLE</b>	<b>8-7</b>
<b>8.6.1</b>	<b>Defining Control Tables</b> _____	<b>8-7</b>
<b>8.6.2</b>	<b>Inserting Table Information</b> _____	<b>8-8</b>
<b>8.6.3</b>	<b>Formatting Information for Output</b> _____	<b>8-10</b>
<b>8.7</b>	<b>HOW TO LINK TO THE CROSS-REFERENCE SHAREABLE IMAGE</b>	<b>8-11</b>

## LIB\$ REFERENCE SECTION

LIB\$ADAWI	LIB-3
LIB\$ADD_TIMES	LIB-5
LIB\$ADDX	LIB-7
LIB\$ANALYZE_SDESC	LIB-10
LIB\$ASN_WTH_MBX	LIB-12
LIB\$AST_IN_PROG	LIB-15
LIB\$ATTACH	LIB-17
LIB\$BBCCI	LIB-19
LIB\$BBSSI	LIB-21
LIB\$CALLG	LIB-23
LIB\$CHAR	LIB-25
LIB\$CONVERT_DATE_STRING	LIB-27
LIB\$CRC	LIB-31
LIB\$CRC_TABLE	LIB-33
LIB\$CREATE_DIR	LIB-36
LIB\$CREATE_USER_VM_ZONE	LIB-40
LIB\$CREATE_VM_ZONE	LIB-44

## Contents

LIB\$CRF_INS_KEY	LIB-50
LIB\$CRF_INS_REF	LIB-52
LIB\$CRF_OUTPUT	LIB-55
LIB\$CURRENCY	LIB-59
LIB\$CVT_DX_DX	LIB-61
LIB\$CVT_FROM_INTERNAL_TIME	LIB-67
LIB\$CVTF_FROM_INTERNAL_TIME	LIB-70
LIB\$CVT_TO_INTERNAL_TIME	LIB-72
LIB\$CVTF_TO_INTERNAL_TIME	LIB-74
LIB\$CVT_XTB	LIB-76
LIB\$CVT_VECTIM	LIB-78
LIB\$DATE_TIME	LIB-80
LIB\$DAY	LIB-82
LIB\$DAY_OF_WEEK	LIB-84
LIB\$DECODE_FAULT	LIB-86
LIB\$DEC_OVER	LIB-104
LIB\$DELETE_FILE	LIB-106
LIB\$DELETE_LOGICAL	LIB-114
LIB\$DELETE_SYMBOL	LIB-116
LIB\$DELETE_VM_ZONE	LIB-118
LIB\$DIGIT_SEP	LIB-120
LIB\$DISABLE_CTRL	LIB-122
LIB\$DO_COMMAND	LIB-124
LIB\$DIV	LIB-126
LIB\$EMODD	LIB-128
LIB\$EMODF	LIB-130
LIB\$EMODG	LIB-132
LIB\$EMODH	LIB-134
LIB\$EMUL	LIB-136
LIB\$ENABLE_CTRL	LIB-138
LIB\$ESTABLISH	LIB-140
LIB\$EXTV	LIB-142
LIB\$EXTZV	LIB-145
LIB\$FFX	LIB-147
LIB\$FID_TO_NAME	LIB-149
LIB\$FILE_SCAN	LIB-151
LIB\$FILE_SCAN_END	LIB-153
LIB\$FIND_FILE	LIB-155
LIB\$FIND_FILE_END	LIB-159
LIB\$FIND_IMAGE_SYMBOL	LIB-160
LIB\$FIND_VM_ZONE	LIB-163
LIB\$FIXUP_FLT	LIB-165

LIB\$FLT_UNDER	LIB-167
LIB\$FORMAT_DATE_TIME	LIB-169
LIB\$FREE_DATE_TIME_CONTEXT	LIB-172
LIB\$FREE_EF	LIB-174
LIB\$FREE_LUN	LIB-175
LIB\$FREE_TIMER	LIB-176
LIB\$FREE_VM	LIB-177
LIB\$FREE_VM_PAGE	LIB-179
LIB\$GETDVI	LIB-181
LIB\$GETJPI	LIB-186
LIB\$GETQUI	LIB-191
LIB\$GETSYI	LIB-196
LIB\$GET_COMMAND	LIB-199
LIB\$GET_COMMON	LIB-202
LIB\$GET_DATE_FORMAT	LIB-204
LIB\$GET_EF	LIB-206
LIB\$GET_FOREIGN	LIB-208
LIB\$GET_INPUT	LIB-212
LIB\$GET_LUN	LIB-215
LIB\$GET_MAXIMUM_DATE_LENGTH	LIB-216
LIB\$GET_SYMBOL	LIB-219
LIB\$GET_USERS_LANGUAGE	LIB-222
LIB\$GET_VM	LIB-223
LIB\$GET_VM_PAGE	LIB-225
LIB\$ICHAR	LIB-227
LIB\$INDEX	LIB-229
LIB\$INIT_DATE_TIME_CONTEXT	LIB-231
LIB\$INIT_TIMER	LIB-235
LIB\$INSERT_TREE	LIB-237
LIB\$INSQHI	LIB-248
LIB\$INSQTI	LIB-251
LIB\$INSV	LIB-253
LIB\$INT_OVER	LIB-255
LIB\$LEN	LIB-257
LIB\$LOCC	LIB-258
LIB\$LOOKUP_KEY	LIB-261
LIB\$LOOKUP_TREE	LIB-265
LIB\$LP_LINES	LIB-267
LIB\$MATCHC	LIB-270
LIB\$MATCH_COND	LIB-272
LIB\$MOVC3	LIB-275
LIB\$MOVC5	LIB-276

## Contents

LIB\$MOVTC	LIB-278
LIB\$MOVTUC	LIB-295
LIB\$MULT_DELTA_TIME	LIB-297
LIB\$MULTF_DELTA_TIME	LIB-298
LIB\$PAUSE	LIB-299
LIB\$POLYD	LIB-300
LIB\$POLYF	LIB-302
LIB\$POLYG	LIB-305
LIB\$POLYH	LIB-307
LIB\$PUT_COMMON	LIB-309
LIB\$PUT_OUTPUT	LIB-311
LIB\$RADIX_POINT	LIB-313
LIB\$REMQHI	LIB-315
LIB\$REMQTI	LIB-317
LIB\$RENAME_FILE	LIB-319
LIB\$RESERVE_EF	LIB-327
LIB\$RESET_VM_ZONE	LIB-329
LIB\$REVERT	LIB-331
LIB\$RUN_PROGRAM	LIB-332
LIB\$SCANC	LIB-334
LIB\$SCOPY_DXDX	LIB-336
LIB\$SCOPY_R_DX	LIB-338
LIB\$SET_LOGICAL	LIB-340
LIB\$SET_SYMBOL	LIB-343
LIB\$FREE1_DD	LIB-347
LIB\$SFREEN_DD	LIB-348
LIB\$SGET1_DD	LIB-350
LIB\$SHOW_TIMER	LIB-352
LIB\$SHOW_VM	LIB-356
LIB\$SHOW_VM_ZONE	LIB-359
LIB\$SIGNAL	LIB-365
LIB\$SIG_TO_RET	LIB-369
LIB\$SIG_TO_STOP	LIB-372
LIB\$SIM_TRAP	LIB-374
LIB\$SKPC	LIB-376
LIB\$SPANC	LIB-378
LIB\$SPAWN	LIB-382
LIB\$STAT_TIMER	LIB-388
LIB\$STAT_VM	LIB-392
LIB\$STOP	LIB-394
LIB\$SUB_TIMES	LIB-397
LIB\$SUBX	LIB-399

LIB\$\$SYS_ASCTIM	LIB-401
LIB\$\$SYS_FAO	LIB-404
LIB\$\$SYS_FAOL	LIB-406
LIB\$\$SYS_GETMSG	LIB-408
LIB\$TPARSE	LIB-411
LIB\$TRA_ASC_EBC	LIB-453
LIB\$TRA_EBC_ASC	LIB-457
LIB\$TRAVERSE_TREE	LIB-459
LIB\$TRIM_FILESPEC	LIB-461
LIB\$VERIFY_VM_ZONE	LIB-464
LIB\$WAIT	LIB-465

---

**INDEX**

---

**EXAMPLES**

5-1	Monitoring Heap Operations with a User-Defined Zone	5-17
-----	---	------

---

**FIGURES**

2-1	Variable-Length Bit Field	2-11
4-1	Format of the Condition Value	4-6
4-2	Sample Stack Scan for Condition Handlers	4-9
4-3	Format of the Signal Argument Vector	4-10
4-4	Signal Argument Vector for the Reserved Operand Error Conditions	4-11
4-5	Signal Argument Vector for RTL Mathematics Routine Errors	4-11
4-6	Format of a Mechanism Argument Vector	4-12
4-7	Formats of Message Sequences	4-17
4-8	Stack After Second Exception Condition Is Signaled	4-19
4-9	Arguments Passed to Condition Handler During Unwind	4-24
4-10	Using a Condition Handler to Log an Error Message	4-28
5-1	Virtual Address Overview	5-2
5-2	Hierarchy of Memory Management Routines	5-4
5-3	Memory Fragmentation	5-6
5-4	Boundary Tags	5-9
7-1	Sequence of Events During Image Initialization	7-3
8-1	Using Cross-Reference Routines	8-2

## Contents

8-2	Summary of Symbol Names and Values _____	8-5
8-3	Summary of Symbol Names, Values, and Name of Referring Modules _____	8-5
8-4	Summary Indicating Defining Module _____	8-6
LIB-1	Structure of a Protection Mask _____	LIB-37
LIB-2	Summary of Symbol Names and Values _____	LIB-57
LIB-3	Summary of Symbol Names, Values, and Name of Referring Modules _____	LIB-57
LIB-4	Summary Indicating Defining Module _____	LIB-58
LIB-5	Keyword Table _____	LIB-263
LIB-6	LIB\$TPARSE Argument Block _____	LIB-412
LIB-7	Transition Diagram for the Mythical Utility _____	LIB-425
LIB-8	Diagram of the Mythical Utility _____	LIB-426

---

## TABLES

1-1	LIB\$ Routines _____	1-1
2-1	System Service Access Routines _____	2-2
2-2	CLI Access Routines _____	2-2
2-3	Variable-Length Bit Field Routines _____	2-10
2-4	Integer and Floating-Point Routines _____	2-12
2-5	Queue Access Routines _____	2-13
2-6	Character String Routines _____	2-15
2-7	Miscellaneous Instruction Routines _____	2-16
2-8	Processwide Resource Allocation Routines _____	2-17
2-9	Performance Measurement Routines _____	2-18
2-10	The Code Argument in LIB\$SHOW_TIMER and LIB\$STAT_TIMER _____	2-19
2-11	Routines for Customizing Output _____	2-21
2-12	Miscellaneous Interface Routines _____	2-22
3-1	Date/Time Formatting Routines _____	3-1
3-2	Input String Punctuation and Defaults _____	3-9
3-3	Predefined Output Date Formats _____	3-10
3-4	Predefined Output Time Formats _____	3-11
3-5	Legible Format Mnemonics _____	3-12
4-1	Condition Handling and Signaling Routines _____	4-1
4-2	Interaction Between Handlers and Default Handlers _____	4-15
5-1	Overhead for Area Control Blocks _____	5-10
5-2	Possible Values for the Block Size Attribute _____	5-11
5-3	Attribute Values for the Default Zone _____	5-12

5-4	Allocation Algorithms _____	5-15
8-1	Cross-Reference Routines _____	8-1
LIB-1	Acceptable Subset of VAX Standard Data Types _____	LIB-63
LIB-2	Data Types Accepted by LIB\$CVT_DX_DX _____	LIB-64
LIB-3	Destination NBDS Formats _____	LIB-65
LIB-4	Formats Used for LIB\$GETDVI Strings _____	LIB-184
LIB-5	Item Code Formats for LIB\$GETJPI _____	LIB-188
LIB-6	Item Code Formats for LIB\$GETQUI _____	LIB-194
LIB-7	LIB\$AB_ASC_EBC _____	LIB-280
LIB-8	LIB\$AB_ASC_EBC_REV _____	LIB-281
LIB-9	LIB\$AB_EBC_ASC _____	LIB-282
LIB-10	LIB\$AB_EBC_ASC_REV _____	LIB-283
LIB-11	LIB\$AB_CVTPT_O _____	LIB-284
LIB-12	LIB\$AB_CVTPT_U _____	LIB-285
LIB-13	LIB\$AB_CVTTP_O _____	LIB-286
LIB-14	LIB\$AB_CVTTP_U _____	LIB-287
LIB-15	LIB\$AB_CVT_O_U _____	LIB-288
LIB-16	LIB\$AB_CVT_U_O _____	LIB-288
LIB-17	LIB\$AB_CVTPT_Z _____	LIB-289
LIB-18	LIB\$AB_CVTTP_Z _____	LIB-290
LIB-19	LIB\$AB_UPCASE _____	LIB-291
LIB-20	LIB\$AB_LOWERCASE _____	LIB-292
LIB-21	The Alphabet of LIB\$TPARSE _____	LIB-414
LIB-22	Argument Block Fields _____	LIB-418
LIB-23	LIB\$AB_ASC_EBC _____	LIB-454
LIB-24	LIB\$AB_EBC_ASC _____	LIB-458





---

## Preface

This manual provides users of the VMS operating system with detailed usage and reference information on library routines supplied in the LIB\$ facility of the Run-Time Library.

Run-Time Library routines can only be used in programs written in languages that produce native code for the VAX hardware. At present, these languages include VAX MACRO and the following compiled high-level languages:

- VAX Ada
- VAX BASIC
- VAX BLISS-32
- VAX C
- VAX COBOL
- VAX COBOL-74
- VAX CORAL
- VAX DIBOL
- VAX FORTRAN
- VAX Pascal
- VAX PL/I
- VAX RPG
- VAX SCAN

Interpreted languages that can also access Run-Time Library routines include VAX DSM and DATATRIEVE.

---

## Intended Audience

This manual is intended for system and application programmers who want to call Run-Time Library routines.

---

## Document Structure

This manual is organized into two parts as follows:

- The introductory chapters provide guidelines and reference material on specific types of library routines. The material is covered as follows:
  - Chapter 1 provides a brief overview of the LIB\$ facility and lists the LIB\$ routines and their functions.
  - Chapter 2 provides an overview and reference guide for routines that are used to access VMS system components.
  - Chapter 3 describes the international date/time input and output routines.
  - Chapter 4 provides detailed information on the condition handling routines.
  - Chapter 5 explains process-wide resource allocation routines, specifically the allocation and deallocation of virtual memory and the use of virtual memory zones.
  - Chapter 6 explains some methods and aids for debugging programs that use virtual memory zones.

## Preface

Chapter 7 discusses image initialization and termination, with special emphasis on the use of LIB\$INITIALIZE.

Chapter 8 discusses use of the cross-reference routines.

- The LIB\$ Reference Section describes each library routine contained in the LIB\$ facility of the Run-Time Library. This information is presented using the documentation format described in the *Introduction to the VMS Run-Time Library*. Routine descriptions appear in alphabetical order by routine name.

---

## Associated Documents

The Run-Time Library routines are documented in a series of reference manuals. A general overview of the Run-Time Library and a description of how the Run-Time Library routines are accessed are presented in the *Introduction to the VMS Run-Time Library*. Descriptions of the other RTL facilities and their corresponding routines and usages are discussed in the following books:

- *The VMS RTL DECTalk (DTK\$) Manual*
- *The VMS RTL Mathematics (MTH\$) Manual*
- *The VMS RTL General Purpose (OTS\$) Manual*
- *The VMS RTL Parallel Processing (PPL\$) Manual*
- *The VMS RTL Screen Management (SMG\$) Manual*
- *The VMS RTL String Manipulation (STR\$) Manual*

The VAX Procedure Calling and Condition Handling Standard, which is documented in the *Introduction to VMS System Routines*, contains useful information for anyone who wants to call Run-Time Library routines.

Application programmers in any language can refer to the *Guide to Creating VMS Modular Procedures* for the Modular Programming Standard and other guidelines.

High-level language programmers will find additional information on calling Run-Time Library routines in their language reference manual. Additional information may also be found in the language user's guide provided with your VAX language software.

The *Guide to Using VMS Command Procedures* may also be useful.

For a complete list and description of the manuals in the VMS documentation set, see the *Overview of VMS Documentation*.

---

**Conventions**

Convention	Meaning
<span style="border: 1px solid black; padding: 2px;">RET</span>	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
\$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
\$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

Other conventions used in the documentation of Run-Time Library routines are described in the *Introduction to the VMS Run-Time Library*.



---

## New and Changed Features

The following LIB\$ routines have been added to the VMS Run-Time Library for Version 5.0:

---

### New LIB\$ Routines for Version 5.0

---

LIB\$ADAWI	LIB\$FREE_DATE_TIME_CONTEXT
LIB\$ADD_TIMES	LIB\$GETQUI
LIB\$CONVERT_DATE_STRING	LIB\$GET_DATE_FORMAT
LIB\$CVT_FROM_INTERNAL_TIME	LIB\$GET_MAXIMUM_DATE_LENGTH
LIB\$CVTF_FROM_INTERNAL_TIME	LIB\$GET_USERS_LANGUAGE
LIB\$CVT_TO_INTERNAL_TIME	LIB\$INIT_DATE_TIME_CONTEXT
LIB\$CVTF_TO_INTERNAL_TIME	LIB\$MULT_DELTA_TIME
LIB\$CVT_VECTIM	LIB\$MULTF_DELTA_TIME
LIB\$FID_TO_NAME	LIB\$SHOW_VM_ZONE
LIB\$FIND_VM_ZONE	LIB\$SUB_TIMES
LIB\$FORMAT_DATE_TIME	LIB\$VERIFY_VM_ZONE

---



# 1

## Overview of the LIB\$ Facility

---

This manual discusses the Run-Time Library LIB\$ routines that perform general purpose (library) functions. One of the functions of the LIB\$ facility is to provide a callable interface to components of VMS that are difficult to use in a high-level language. LIB\$ routines allow access to the following:

- System Services
- The Command Language Interpreter (CLI)
- Some VAX machine instructions

In addition, LIB\$ routines allow you to perform the following operations:

- Allocate the resources that your process needs, such as virtual memory and event flags
- Convert data types for I/O
- Enable detection of hardware exceptions
- Establish condition handlers
- Generate and display timing statistics while your program is running
- Get and put strings in the process common storage area
- Obtain records from devices
- Obtain the system date and time in various formats
- Process cross-reference data
- Search for specified files
- Set up and use binary trees
- Signal exceptions

The following table contains all of the LIB\$ routines and their functions.

**Table 1–1 LIB\$ Routines**

<b>Routine Name</b>	<b>Function</b>
LIB\$ADAWI	Add adjacent word with interlock
LIB\$ADD_TIMES	Add two quadword times
LIB\$ADDX	Add two multiple-precision binary numbers
LIB\$ANALYZE_SDESC	Analyze a string descriptor
LIB\$ASN_WTH_MBX	Assign a channel to a mailbox
LIB\$AST_IN_PROG	AST in progress
LIB\$ATTACH	Attach a terminal to a process



# Overview of the LIB\$ Facility

**Table 1–1 (Cont.) LIB\$ Routines**

<b>Routine Name</b>	<b>Function</b>
LIB\$BBCCI	Test and clear a bit with interlock
LIB\$BBSSI	Test and set a bit with interlock
LIB\$CALLG	Call a procedure with a general argument list
LIB\$CHAR	Transform a byte to the first character of a string
LIB\$CONVERT_DATE_STRING	Convert a date string to a quadword
LIB\$CRC	Calculate a Cyclic Redundancy Check (CRC)
LIB\$CRC_TABLE	Construct a Cyclic Redundancy Check (CRC) table
LIB\$CREATE_DIR	Create a directory
LIB\$CREATE_USER_VM_ZONE	Create a user-defined storage zone
LIB\$CREATE_VM_ZONE	Create a new storage zone
LIB\$CRF_INS_KEY	Insert a key in the cross-reference table
LIB\$CRF_INS_REF	Insert a reference to a key in the cross-reference table
LIB\$CRF_OUTPUT	Output some cross-reference table information
LIB\$CURRENCY	Get the system currency symbol
LIB\$CVT_DX_DX	Convert the specified data type
LIB\$CVT_FROM_INTERNAL_TIME	Convert internal time to external time
LIB\$CVTF_FROM_INTERNAL_TIME	Convert internal time to external time (F-floating value)
LIB\$CVT_TO_INTERNAL_TIME	Convert external time to internal time
LIB\$CVTF_TO_INTERNAL_TIME	Convert external time to internal time (F-floating value)
LIB\$CVT_xTB	Convert numeric text to binary
LIB\$CVT_VECTIM	Convert 7-word vector to internal time
LIB\$DATE_TIME	Return the date and time as a string
LIB\$DAY	Return the day number as a longword integer
LIB\$DAY_OF_WEEK	Return the numeric day of the week
LIB\$DECODE_FAULT	Decode instruction stream during a fault
LIB\$DEC_OVER	Enable or disable decimal overflow detection
LIB\$DELETE_FILE	Delete one or more files
LIB\$DELETE_LOGICAL	Delete a logical name
LIB\$DELETE_SYMBOL	Delete a CLI symbol

# Overview of the LIB\$ Facility

**Table 1–1 (Cont.) LIB\$ Routines**

<b>Routine Name</b>	<b>Function</b>
LIB\$DELETE_VM_ZONE	Delete a virtual memory zone
LIB\$DIGIT_SEP	Get the digit separator symbol
LIB\$DISABLE_CTRL	Disable CLI interception of control characters
LIB\$DO_COMMAND	Execute the specified command
LIB\$EDIV	Perform an extended-precision divide
LIB\$EMODD	Perform extended multiply and integerize for D-floating values
LIB\$EMODF	Perform extended multiply and integerize for F-floating values
LIB\$EMODG	Perform extended multiply and integerize for G-floating values
LIB\$EMODH	Perform extended multiply and integerize for H-floating values
LIB\$EMUL	Perform an extended-precision multiply
LIB\$ENABLE_CTRL	Enable CLI interception of control characters
LIB\$ESTABLISH	Establish a condition handler
LIB\$EXTV	Extract a field and sign-extend
LIB\$EXTZV	Extract a zero-extended field
LIB\$FFx	Find the first clear or set bit
LIB\$FID_TO_NAME	Convert a device and file ID to a file specification
LIB\$FILE_SCAN	Perform a file scan
LIB\$FILE_SCAN_END	End of file scan
LIB\$FIND_FILE	Find a file
LIB\$FIND_FILE_END	End of find file
LIB\$FIND_IMAGE_SYMBOL	Merge activate an image symbol
LIB\$FIND_VM_ZONE	Find the next valid zone
LIB\$FIXUP_FLT	Fix floating reserved operand
LIB\$FLT_UNDER	Floating-point underflow detection
LIB\$FORMAT_DATE_TIME	Format a date and/or time
LIB\$FREE_DATE_TIME_CONTEXT	Free the context used to format a date or time
LIB\$FREE_EF	Free an event flag
LIB\$FREE_LUN	Free a logical unit number
LIB\$FREE_TIMER	Free timer storage
LIB\$FREE_VM	Free virtual memory from the program region
LIB\$FREE_VM_PAGE	Free a virtual memory page

# Overview of the LIB\$ Facility

**Table 1–1 (Cont.) LIB\$ Routines**

<b>Routine Name</b>	<b>Function</b>
LIB\$GETDVI	Get device/volume information
LIB\$GETJPI	Get job/process information
LIB\$GETQUI	Get queue information
LIB\$GETSYI	Get systemwide information
LIB\$GET_COMMAND	Get line from SYS\$COMMAND
LIB\$GET_COMMON	Get string from common area
LIB\$GET_DATE_FORMAT	Return the user's date input format
LIB\$GET_EF	Get an event flag
LIB\$GET_FOREIGN	Get foreign command line
LIB\$GET_INPUT	Get line from SYS\$INPUT
LIB\$GET_LUN	Get logical unit number
LIB\$GET_MAXIMUM_DATE_LENGTH	Get the maximum possible date/time string length
LIB\$GET_SYMBOL	Get the value of a CLI symbol
LIB\$GET_USERS_LANGUAGE	Return the user's language choice
LIB\$GET_VM	Allocate virtual memory
LIB\$GET_VM_PAGE	Get a virtual memory page
LIB\$ICHAR	Convert first character of string to integer
LIB\$INDEX	Index to relative position of substring
LIB\$INIT_DATE_TIME_CONTEXT	Initialize the context used in formatting date/time strings
LIB\$INIT_TIMER	Initialize times and counts
LIB\$INSERT_TREE	Insert entry in a balanced binary tree
LIB\$INSQHI	Insert entry at the head of a queue
LIB\$INSQTI	Insert entry at the tail of a queue
LIB\$INSV	Insert a variable bit field
LIB\$INT_OVER	Integer overflow detection
LIB\$LEN	Return the length of a string as a longword
LIB\$LOCC	Locate a character
LIB\$LOOKUP_KEY	Look up keyword in table
LIB\$LOOKUP_TREE	Look up an entry in a balanced binary tree
LIB\$LP_LINES	Specify the number of lines on each printer page
LIB\$MATCHC	Match characters, return relative position
LIB\$MATCH_COND	Match condition values
LIB\$MOVC3	Move characters

# Overview of the LIB\$ Facility

**Table 1–1 (Cont.) LIB\$ Routines**

<b>Routine Name</b>	<b>Function</b>
LIB\$MOVC5	Move characters with fill
LIB\$MOVTC	Move translated characters
LIB\$MOVTUC	Move translated until character
LIB\$MULT_DELTA_TIME	Multiply delta time by scalar
LIB\$MULTF_DELTA_TIME	Multiply delta time by F-floating scalar
LIB\$PAUSE	Pause program execution
LIB\$POLYD	Evaluate polynomials for D-floating values
LIB\$POLYF	Evaluate polynomials for F-floating values
LIB\$POLYG	Evaluate polynomials for G-floating values
LIB\$POLYH	Evaluate polynomials for H-floating values
LIB\$PUT_COMMON	Put string into common area
LIB\$PUT_OUTPUT	Put line to SY\$\$OUTPUT
LIB\$RADIX_POINT	Radix point symbol
LIB\$REMQHI	Remove entry from head of queue
LIB\$REMQTI	Remove entry from tail of queue
LIB\$RENAME_FILE	Rename one or more files
LIB\$RESERVE_EF	Reserve an event flag
LIB\$RESET_VM_ZONE	Reset virtual memory zone
LIB\$REVERT	Revert to the handler of the procedure activator
LIB\$RUN_PROGRAM	Run new program
LIB\$SCANC	Scan for characters and return relative position
LIB\$SCOPY_DXDX	Copy source string by descriptor to destination
LIB\$SCOPY_R_DX	Copy source string by reference to destination
LIB\$SET_LOGICAL	Set logical name
LIB\$SET_SYMBOL	Set value of a CLI symbol
LIB\$SFREE1_DD	Free one or more dynamic strings
LIB\$SFREEN_DD	Free n dynamic strings
LIB\$SGET1_DD	Get one dynamic string
LIB\$SHOW_TIMER	Show accumulated times and counts
LIB\$SHOW_VM	Show virtual memory statistics
LIB\$SHOW_VM_ZONE	Display information about a virtual memory zone

# Overview of the LIB\$ Facility

**Table 1–1 (Cont.) LIB\$ Routines**

<b>Routine Name</b>	<b>Function</b>
LIB\$SIGNAL	Signal exception condition
LIB\$SIG_TO_RET	Convert signaled message to a return status
LIB\$SIG_TO_STOP	Convert a signaled condition to a signaled stop
LIB\$SIM_TRAP	Simulate floating trap
LIB\$SKPC	Skip equal characters
LIB\$SPANC	Skip selected characters
LIB\$SPAWN	Spawn a subprocess
LIB\$STAT_TIMER	Return accumulated time and count statistics
LIB\$STAT_VM	Return virtual memory statistics
LIB\$STOP	Stop execution and signal the condition
LIB\$SUB_TIMES	Subtract two quadword times
LIB\$SUBX	Perform multiple-precision binary subtraction
LIB\$SYS_ASCTIM	Invoke \$ASCTIM to convert binary time to ASCII
LIB\$SYS_FAO	Invoke \$FAO system service to format output
LIB\$SYS_FAOL	Invoke \$FAOL system service to format output
LIB\$SYS_GETMSG	Invoke \$GETMSG system service to get message text
LIB\$TPARSE	Implement a table-driven, finite-state parser
LIB\$TRA_ASC_EBC	Translate ASCII to EBCDIC
LIB\$TRA_EBC_ASC	Translate EBCDIC to ASCII
LIB\$TRAVERSE_TREE	Traverse a balanced binary tree
LIB\$TRIM_FILESPEC	Fit long file specification into fixed field
LIB\$VERIFY_VM_ZONE	Verify a virtual memory zone
LIB\$WAIT	Wait a specified period of time

# 2

---

## Access to VMS System Components

Run-Time Library LIB\$ routines allow access to the following VMS system components:

- System services
- The Command Language Interpreter
- Some VAX machine instructions

This chapter discusses in detail how you can access VMS system components using LIB\$ routines.

---

### 2.1 System Service Access Routines

You can usually call VMS system services directly from your program. However, system services return only fixed-length strings. In some applications, you may want the result of a system service to be returned as a character array, dynamic string, or variable-length string. For this reason, the Run-Time Library provides *jacket* routines for the system services that return strings.

You call these routines exactly as you would the corresponding system service, but you can pass an output argument of any valid string class. The routines write the output string using the semantics (fixed, varying, or dynamic) associated with the string's descriptor.

The jacket routines follow the conventions established for all Run-Time Library routines, except that the arguments are listed in the order of the arguments for the corresponding system service. Thus, they may not be listed in the standard Run-Time Library order (read, modify, write).

For example, LIB\$SYS\_ASCTIM calls the system service \$ASCTIM to convert a binary date and time value to ASCII text. It returns the resulting string using the semantics that the calling program specifies in the destination string argument.

For further information about the operations of the system services, see the *VMS System Services Reference Manual*.

The Run-Time Library routines provide access only to the system services that produce output strings, listed in Table 2-1. The corresponding Run-Time Library routines recognize all VAX string classes.

The Run-Time Library does not provide jacket routines for all system services that accept strings as input. Your program should pass only fixed-length or dynamic input strings to all system services and Run-Time Library jacket routines.

# Access to VMS System Components

## 2.1 System Service Access Routines

**Table 2-1 System Service Access Routines**

Entry Point	System Service	Function
LIB\$SYS_ASCTIM	\$ASCTIM	Converts system time in binary form to ASCII text
LIB\$SYS_FAO	\$FAO	Converts a binary value to ASCII text
LIB\$SYS_FAOL	\$FAOL	Converts a binary value to ASCII text, using a list argument
LIB\$SYS_GETMSG	\$GETMSG	Obtains a system or user-defined message text
LIB\$SYS_TRNLOG	\$TRNLOG	Returns the translation of the specified logical name

## 2.2 Access to the Command Language Interpreter

Two Command Language Interpreters (CLIs) are available under VMS: DCL and MCR. The Run-Time Library provides several routines that provide access to the VMS CLI callback facility. These routines allow your program to call the current CLI. In most cases, these routines are called from programs that execute as part of a command procedure. They allow the command procedure and the CLI to exchange information.

These routines call the CLI associated with the current process to perform the specified function. In some cases, however, there is no CLI present. For example, the program may be running directly as a subprocess or as a detached process. If no CLI is present, these routines return the status LIB\$\_NOCLI. Therefore, you should be sure that these routines are called when a CLI is active. Table 2-2 lists the Run-Time Library routines that access the Command Language Interpreter.

**Table 2-2 CLI Access Routines**

Entry Point	Function
LIB\$GET_FOREIGN	Gets a command line
LIB\$DO_COMMAND	Executes a command line after exiting the current program
LIB\$RUN_PROGRAM	Runs another program after exiting the current program (chain)
LIB\$GET_SYMBOL	Returns the value of a CLI symbol as a string
LIB\$DELETE_SYMBOL	Deletes a CLI symbol
LIB\$SET_SYMBOL	Defines or redefines a CLI symbol
LIB\$DELETE_LOGICAL	Deletes a supervisor-mode process logical name
LIB\$SET_LOGICAL	Defines or redefines a supervisor-mode process logical name
LIB\$DISABLE_CTRL	Disables CLI interception of control characters

# Access to VMS System Components

## 2.2 Access to the Command Language Interpreter

**Table 2–2 (Cont.) CLI Access Routines**

Entry Point	Function
LIB\$ENABLE_CTRL	Enables CLI interception of control characters
LIB\$ATTACH	Attaches a terminal to another process
LIB\$SPAWN	Creates a subprocess of the current process

The following routines execute only when the current Command Language Interpreter is DCL:

LIB\$GET\_SYMBOL  
LIB\$SET\_SYMBOL  
LIB\$DELETE\_SYMBOL  
LIB\$DISABLE\_CTRL  
LIB\$ENABLE\_CTRL  
LIB\$SPAWN  
LIB\$ATTACH

### 2.2.1 Obtaining the Command Line

LIB\$GET\_FOREIGN returns the contents of the command line that you use to activate an image. It can be used to give your program access to the qualifiers of a foreign command or to prompt for further command line text.

A *foreign command* is a command that you can define and then use as if it were a DCL or MCR command in order to run a program. When you use the foreign command at command level, the Command Language Interpreter parses the foreign command only and activates the image. It ignores any options or qualifiers that you have defined for the foreign command. Once the CLI has activated the image, the program can call LIB\$GET\_FOREIGN to obtain and parse the remainder of the command line (after the command itself) for whatever options it may contain.

The *VMS DCL Dictionary* shows how to define a foreign command.

The action of LIB\$GET\_FOREIGN depends on the environment in which the image is activated:

- If you use a foreign command to invoke the image, you can call LIB\$GET\_FOREIGN to obtain the command qualifiers following the foreign command. You can also use LIB\$GET\_FOREIGN to prompt repeatedly for more qualifiers after the command. This technique is illustrated in the example following this list.
- If the image is in the SYS\$SYSTEM: directory, the image can be invoked by the DCL MCR command or by the MCR Command Language Interpreter. In this case, LIB\$GET\_FOREIGN returns the command line text following the image name.
- If the image is invoked by a DCL RUN command, LIB\$GET\_FOREIGN can be used to prompt for additional text.
- If the image is not invoked by a foreign command or MCR, or if there is no information remaining on the command line, and the user-supplied prompt is present, LIB\$GET\_INPUT is called to prompt for a command line. If the prompt is not present, LIB\$GET\_FOREIGN returns a zero length string.



# Access to VMS System Components

## 2.2 Access to the Command Language Interpreter

### Example

The following PL/I example illustrates the use of the optional **force-prompt** argument to permit repeated calls to LIB\$GET\_FOREIGN. The command line text will be retrieved on the first pass only; after this, the program will prompt from SYS\$INPUT.

```
EXAMPLE: ROUTINE OPTIONS (MAIN);
%INCLUDE $STSDEF;          /* Status-testing definitions */
DECLARE COMMAND_LINE CHARACTER(80) VARYING,
        PROMPT_FLAG FIXED BINARY(31) INIT(0),
        LIB$GET_FOREIGN ENTRY (CHARACTER(*) VARYING,
                                CHARACTER(*) VARYING,
                                FIXED BINARY(15),
                                FIXED BINARY(31))
                                OPTIONS(VARIABLE) RETURNS (FIXED BINARY(31)),
        RMS$_EOF GLOBALREF FIXED BINARY(31) VALUE;

/* Call LIB$GET_FOREIGN repeatedly to obtain and print
   subcommand text. Exit when end-of-file is found. */
DO WHILE ('1'B);          /* Do while TRUE */
    STS$VALUE = LIB$GET_FOREIGN
                (COMMAND_LINE,'Input: ',,
                 PROMPT_FLAG);
    IF STS$SUCCESS THEN
        PUT LIST (' Command was ',COMMAND_LINE);
    ELSE DO;
        IF STS$VALUE ^= RMS$_EOF THEN
            PUT LIST ('Error encountered');
        RETURN;
    END;
    PUT SKIP;              /* Skip to next line */
END;                       /* End of DO WHILE loop */
END;
```

Assuming that this program is present as SYS\$SYSTEM:EXAMPLE.EXE, you can define the foreign command EXAMPLE to invoke it.

```
$ EXAM*PLE := $EXAMPLE
```

Note the optional use of the asterisk in the symbol name to denote an abbreviated command name. This permits the command name to be abbreviated as EXAM, EXAMP, EXAMPL, or fully specified as EXAMPLE. See the *VMS DCL Dictionary* for information on abbreviated command names.

Note also the use of the dollar sign before the image name. This is necessary in foreign commands.

Now assume that a user runs the image by typing the foreign command and giving "subcommands," which the program displays. Text typed by the user is in red.

```
$ EXAMP Subcommand 1
  Command was      SUBCOMMAND 1
Input: Subcommand 2
  Command was      SUBCOMMAND 2
Input: ^Z
$
```

# Access to VMS System Components

## 2.2 Access to the Command Language Interpreter

**Subcommand 1** was obtained from the command line; the program prompts the user for the second subcommand. The program terminated when the user pressed CTRL/Z (echoed as ^Z) to indicate end-of-file.

### 2.2.2 Chaining from One Program to Another

LIB\$RUN\_PROGRAM causes the current image to exit at the point of the call and directs the Command Language Interpreter, if one is present, to start running another program. If LIB\$RUN\_PROGRAM executes successfully, control passes to the second program; if not, control passes to the Command Language Interpreter. The calling program cannot regain control. This technique is called *chaining*.

This routine is provided primarily for compatibility with PDP-11 systems, where chaining is used to extend the address space of a system. It may also be useful in a VMS environment where address space is severely limited and large images are not possible. For example, you might use chaining to perform system generation (SYSGEN) on a small virtual address space, because of a lack of disk space.

With LIB\$RUN\_PROGRAM, the calling program can pass arguments to the next program in the chain only by using the common storage area. One way to do this is to direct the calling program to call LIB\$PUT\_COMMON to pass the information into the common area. Then the called program calls LIB\$GET\_COMMON to retrieve the data.

In general, this practice is not recommended. There is no convenient way to specify the order and type of arguments passed into the common area, so programs that pass arguments in this way must know about the format of the data before it is passed. FORTRAN COMMON or BASIC MAP/Common areas are global OWN storage. When you use this type of storage, it is very difficult to keep your program modular and AST-reentrant. Further, LIB\$RUN\_PROGRAM cannot be used if no Command Language Interpreter is present, as in the case of image subprocesses and detached subprocesses.

#### Examples

The following PL/I example illustrates the use of LIB\$RUN\_PROGRAM. It prompts the user for the name of a program to run and calls the Run-Time Library routine to execute the specified program.

```
CHAIN: ROUTINE OPTIONS (MAIN) RETURNS (FIXED BINARY (31));
DECLARE LIB$RUN_PROGRAM ENTRY (CHARACTER (*)) /* Address of string
                                           /* descriptor */
                                           RETURNS (FIXED BINARY (31)); /* Return status */
%INCLUDE $STSDEF; /* Include definition of return status values */
DECLARE COMMAND CHARACTER (80);
GET LIST (COMMAND) OPTIONS (PROMPT('Program to run: '));
STS$VALUE = LIB$RUN_PROGRAM (COMMAND);
/*
   If the function call is successful, the program will terminate
   here. Otherwise, return the error status to command level.
*/
RETURN (STS$VALUE);
END CHAIN;
```

# Access to VMS System Components

## 2.2 Access to the Command Language Interpreter

The following COBOL program also demonstrates LIB\$RUN\_PROGRAM. When you compile and link these two programs, the first calls LIB\$RUN\_PROGRAM, which activates the executable image of the second. This call results in the following screen display:

```
THIS MESSAGE DISPLAYED BY PROGRAM PROG2
WHICH WAS RUN BY PROGRAM PROG1
USING LIB$RUN_PROGRAM

IDENTIFICATION DIVISION.
PROGRAM-ID.  PROG1.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01  PROG-NAME  PIC X(9)    VALUE "PROG2.EXE".
01  STAT      PIC 9(9)    COMP.
      88  SUCCESSFUL          VALUE 1.

ROUTINE DIVISION.

001-MAIN.
      CALL "LIB$RUN_PROGRAM"
          USING BY DESCRIPTOR PROG-NAME
          GIVING STAT.
      IF NOT SUCCESSFUL
          DISPLAY "ATTEMPT TO CHAIN UNSUCCESSFUL"
          STOP RUN.

IDENTIFICATION DIVISION.
PROGRAM-ID.  PROG2.

ENVIRONMENT DIVISION.

DATA DIVISION.

ROUTINE DIVISION.

001-MAIN.
      DISPLAY " ".
      DISPLAY "THIS MESSAGE DISPLAYED BY PROGRAM PROG2".
      DISPLAY " ".
      DISPLAY "WHICH WAS RUN BY PROGRAM PROG1".
      DISPLAY " ".
      DISPLAY "USING LIB$RUN_PROGRAM".
      STOP RUN.
```

### 2.2.3 Executing a CLI Command

LIB\$DO\_COMMAND stops program execution and directs the Command Language Interpreter (CLI) to execute a command. The routine's argument is the text of the command line that you want to execute.

This routine is especially useful when you want to execute a CLI command after your program has finished executing. For example, you could set up a series of conditions, each associated with a different command. You could also use the routine to execute a SUBMIT or PRINT command to handle a file that your program creates.

# Access to VMS System Components

## 2.2 Access to the Command Language Interpreter

Because of the following restrictions on LIB\$DO\_COMMAND, you should be careful when you incorporate it in your program.

- After the call to LIB\$DO\_COMMAND, the current image exits, and control cannot return to it.
- The text of the command is passed to the current Command Language Interpreter. Because you can define your own CLI in addition to DCL and MCR, you must make sure that the command will be handled by the intended CLI.
- If the routine is called from a subprocess and no CLI is associated with that subprocess, it will not execute correctly.

You can also use LIB\$DO\_COMMAND to execute a DCL command file. To do this, include the at sign (@) along with a command file specification as the input argument to the routine.

There are also DCL CLI\$ routines that perform the functions of LIB\$DO\_COMMAND. See the *VMS DCL Dictionary* for more information.

### Example

The following PL/I example prompts the user for a DCL command to execute after the program exits:

```
EXECUTE: ROUTINE OPTIONS (MAIN) RETURNS (FIXED BINARY (31));
DECLARE LIB$DO_COMMAND ENTRY (CHARACTER (*)) /* Pass DCL command */
/* by descriptor */
RETURNS (FIXED BINARY (31)); /* Return status */
%INCLUDE $STSDEF; /* Include definition of return status values */
DECLARE COMMAND CHARACTER (80);
GET LIST (COMMAND) OPTIONS (PROMPT('DCL command to execute: '));
STS$VALUE = LIB$DO_COMMAND (COMMAND);
/*
If the call to LIB$DO_COMMAND is successful, the program will terminate
here. Otherwise, it will return the error status to command level.
*/
RETURN (STS$VALUE);
END EXECUTE;
```

This example displays the following prompt:

DCL command to execute:

What you type after this prompt determines the action of LIB\$DO\_COMMAND. LIB\$DO\_COMMAND will execute any command that is entered as a valid string according to the syntax of PL/I. If the command you enter is incomplete, you will be prompted for the rest of the command. For example, if you enter the SHOW command, you will receive the following prompt:

\$\_Show what?:

# Access to VMS System Components

## 2.2 Access to the Command Language Interpreter

### 2.2.4 Using Symbols and Logical Names

The Run-Time Library provides five routines that give you access to the CLI callback facility. These routines allow a program to “call back” to the Command Language Interpreter to perform functions normally performed by CLI commands. These routines perform the following functions:

LIB\$GET_SYMBOL	Returns the value of a CLI symbol as a string. Optionally, this routine also returns the length of the returned value and a value indicating whether the symbol was found in the local or global symbol table. This routine executes only when the current CLI is DCL.
LIB\$SET_SYMBOL	Causes the CLI to define or redefine a CLI symbol. The optional argument specifies whether the symbol is to be defined in the local or global symbol table; the default is local. This routine executes only when the current CLI is DCL.
LIB\$DELETE_SYMBOL	Causes the CLI to delete a symbol. An optional argument specifies the local or global symbol table. If this argument is omitted, the symbol is deleted from the local symbol table. This routine executes only when the current CLI is DCL.
LIB\$SET_LOGICAL	Defines or redefines a supervisor-mode process logical name. Supervisor-mode logical names are not deleted when an image exits. This routine is equivalent to the DCL DEFINE command. LIB\$SET_LOGICAL allows the calling program to define a supervisor-mode process logical name without itself executing in supervisor mode.
LIB\$DELETE_LOGICAL	Deletes a supervisor-mode process logical name. This routine is equivalent to the DCL DEASSIGN command. LIB\$DELETE_LOGICAL does not require the calling program to be executing in supervisor mode to delete a supervisor-mode logical name.

### 2.2.5 Disabling and Enabling Control Characters

Two Run-Time Library routines, LIB\$ENABLE\_CTRL and LIB\$DISABLE\_CTRL, allow you to call the Command Language Interpreter (CLI) to enable or disable control characters. These routines take a longword bit-mask argument that specifies the control character or characters to be disabled or enabled. Acceptable values for this argument are LIB\$M\_CLI\_CTRL and LIB\$M\_CLI\_CTRLT.

# Access to VMS System Components

## 2.2 Access to the Command Language Interpreter

LIB\$DISABLE_CTRL	<p>Disables CLI interception of control characters.</p> <p>This routine performs the same function as the DCL command</p> <pre>SET NOCONTROL = n</pre> <p>where n is equal to T or Y.</p> <p>It prevents the currently active CLI from intercepting the control character specified during an interactive session.</p> <p>For example, you might use LIB\$DISABLE_CTRL to disable CLI interception of CTRL/Y. Normally, CTRL/Y interrupts the current command, command procedure, or image. If LIB\$DISABLE_CTRL is called with LIB\$M_CLI_CTRLY specified as the control character to be disabled, CTRL/Y is treated like CTRL/U followed by a carriage return.</p>
LIB\$ENABLE_CTRL	<p>Enables CLI interception of control characters.</p> <p>This routine performs the same function as the DCL command</p> <pre>SET CONTROL= n,</pre> <p>where n is equal to T or Y. LIB\$ENABLE_CTRL restores the normal operation of CTRL/Y or CTRL/T.</p>

---

### 2.2.6 Creating and Connecting to a Subprocess

You can use LIB\$SPAWN and LIB\$ATTACH together to spawn a subprocess and attach the terminal to that subprocess. These routines will execute correctly only if the current CLI is DCL. For more information on the SPAWN and ATTACH commands, see the *VMS DCL Dictionary*.

LIB\$SPAWN	<p>Spawns a subprocess.</p> <p>This routine is equivalent to the DCL SPAWN command. It requests the CLI to spawn a subprocess for executing CLI commands.</p>
LIB\$ATTACH	<p>Attaches the terminal to another process.</p> <p>This routine is equivalent to the DCL ATTACH command. It requests the CLI to detach the terminal from the current process and reattach it to a different process.</p>

---

## 2.3 Access to VAX Machine Instructions

The VAX instruction set was designed for efficient use by high-level languages, and therefore contains many functions that are directly useful in your programs. However, some of these functions cannot be used directly by high-level languages.

The Run-Time Library provides routines that allow your high-level language program to use most VAX machine instructions that are otherwise unavailable. In most cases, these routines simply execute the instruction, using the arguments you provide. Some routines that accept string

# Access to VMS System Components

## 2.3 Access to VAX Machine Instructions

arguments, however, provide some additional functions that make them easier to use.

These routines fall into the following categories:

- Variable-length bit field instruction routines (Section 2.3.1)
- Integer and floating-point instructions (Section 2.3.2)
- Queue instructions (Section 2.3.3)
- Character string instructions (Section 2.3.4)
- Routine call instructions (Section 2.3.5)
- Cyclic redundancy check instruction (Section 2.3.5)

The *VAX Architecture Reference Manual* describes the VAX instruction set in detail.

### 2.3.1 Variable-Length Bit Field Instruction Routines

The variable-length bit field is a VAX data type used to store small integers packed together in a larger data structure. It is often used to store single flag bits.

The Run-Time Library contains five routines for performing operations on variable-length bit fields. These routines give higher-level languages that do not have the inherent ability to manipulate bit fields direct access to the bit field instructions in the VAX instruction set. Furthermore, if a program calls a routine written in a different language to perform some function that also involves bit manipulation, the called routine can include a call to the Run-Time Library to perform the bit manipulation.

Table 2-3 lists the Run-Time Library variable-length bit field routines.

**Table 2-3 Variable-Length Bit Field Routines**

Entry Point	Function
LIB\$EXTV	Extracts a field from the specified variable-length bit field and returns it in sign-extended longword form.
LIB\$EXTZV	Extracts a field from the specified variable-length bit field and returns it in zero-extended longword form.
LIB\$FFC	Searches the specified field for the first clear bit. If it finds one, it returns SS\$_NORMAL and the bit position ( <b>find-pos</b> ) of the clear bit. If not, it returns a failure status and sets the <b>find-pos</b> argument to the start position plus the size.
LIB\$FFS	Searches the specified field for the first set bit. If it finds one, it returns SS\$_NORMAL and the bit position ( <b>find-pos</b> ) of the set bit. If not, it returns a failure status and sets the <b>find-pos</b> argument to the start position plus the size.
LIB\$INSV	Replaces the specified field with bits zero through [ <b>size</b> - 1] of the source argument ( <b>src</b> ). If the size argument is zero, nothing is inserted.

# Access to VMS System Components

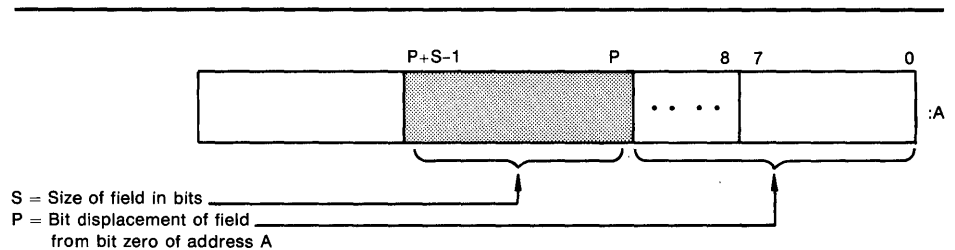
## 2.3 Access to VAX Machine Instructions

Three scalar attributes define a variable bit field:

- *Base address*—the address of the byte in memory that serves as a reference point for locating the bit field.
- *Bit position*—the signed longword containing the displacement of the least significant bit of the field with respect to the bit zero of the base address.
- *Size*—a byte integer indicating the size of the bit field in bits (in the range  $0 \leq \text{size} \leq 32$ ). That is, a bit field can be no more than one longword in length.

Figure 2-1 shows the format of a variable-length bit field. The shaded area indicates the field.

**Figure 2-1 Variable-Length Bit Field**



ZK-1981-84

Bit fields are zero-origin, which means that the routine regards the first bit in the field as being the zero position. For more detailed information about VAX bit numbering and data formats, see the *VAX Architecture Reference Manual*.

The attributes of the bit field are passed to a Run-Time Library routine in the form of three arguments in the order given in the following list:

### **pos**

VMS Usage: longword\_signed  
type: longword integer (signed)  
access: read only  
mechanism: by reference

Bit position relative to the base address. The **pos** argument is the address of a signed longword integer that contains this bit position.

### **size**

VMS Usage: byte\_unsigned  
type: byte (unsigned)  
access: read only  
mechanism: by reference

Size of the bit field. The **size** argument is the address of an unsigned byte which contains this size.

### **base**

VMS Usage: longword\_unsigned  
type: longword (unsigned)  
access: read only  
mechanism: by reference



# Access to VMS System Components

## 2.3 Access to VAX Machine Instructions

Base address. The **base** argument contains the address of the base address.

### Example

The following BASIC example illustrates three Run-Time Library routines. It opens the terminal as a file and specifies "HEX> " as the prompt. This allows you to get input from the terminal without the question mark that VAX BASIC normally adds to the prompt in an INPUT statement. The program calls OTS\$CVT\_TZ\_L to convert the character string input to a longword. It then calls LIB\$EXTZV once for each position in the longword to extract the bit in that position. Because LIB\$EXTVZ is called with a function reference within the PRINT statement, the bits are displayed.

```
10      EXTERNAL LONG FUNCTION
        OTS$CVT_TZ_L,          ! Convert hex text to LONG
        LIB$EXTZV              ! Extract zero-ended bit field

20      OPEN "TT:" FOR INPUT AS FILE #1%      ! Open terminal as a file
        INPUT #1%, "HEX>"; HEXIN$           ! Prompt for input
        STAT%=OTS$CVT_TZ_L(HEXIN$, BINARY%) ! Convert to longword
        IF (STAT% AND 1%) <> 1%              ! Failed?
        THEN
            PRINT "Conversion failed, decimal status ";STAT%
            GO TO 20                          ! Try again
        ELSE
            PRINT HEXIN$,
            PRINT STR$(LIB$EXTZV(N%, 1%, BINARY%));
            FOR N%=31% TO 0% STEP -1%
```

### 2.3.2 Integer and Floating-Point Routines

Integer and floating-point routines give a high-level language program access to the corresponding machine instructions. For a complete description of these instructions, see the *VAX Architecture Reference Manual*. Table 2-4 lists the integer and floating-point routines.

**Table 2-4 Integer and Floating-Point Routines**

Entry Point	Function
LIB\$EMUL	Multiplies integers with extended precision
LIB\$EDIV	Divides integers with extended precision

### 2.3.3 Queue Access Routines

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address. Two longwords, a forward link and a backward link, define the location of the entry in relation to the preceding and succeeding entries. A self-relative queue is a queue in which the links between entries are displacements; the two longwords represent the displacements of the current entry's predecessor and successor. The VAX instructions INSQHI, INSQTI, REMQHI, and REMQTI allow you to insert and remove an entry at the head or tail of a self-relative queue. Each queue instruction has a corresponding Run-Time Library routine.

# Access to VMS System Components

## 2.3 Access to VAX Machine Instructions

The self-relative queue instructions are interlocked and cannot be interrupted, so that other processes cannot insert or remove queue entries while the current program is doing so. Since the operation requires changing two pointers at the same time, a high-level language cannot perform this operation without calling the Run-Time Library queue access routines.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue Access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an asynchronous system trap.

The remove queue instructions (REMQHI or REMQTI) return the address of the removed entry. Some languages, such as BASIC, COBOL, and FORTRAN, do not provide a mechanism for accessing an address returned from a routine. Furthermore, BASIC and COBOL do not allow routines to be arguments.

Table 2-5 lists the Queue Access Routines.

**Table 2-5 Queue Access Routines**

Entry Point	Function
LIB\$INSQHI	Inserts queue entry at head
LIB\$INSQTI	Inserts queue entry at tail
LIB\$REMQHI	Removes queue entry at head
LIB\$REMQTI	Removes queue entry at tail

### Examples

#### LIB\$INSQHI

In BASIC and FORTRAN, queues can be quadword aligned in a named COMMON block by using a linker option file to specify PSECT alignment. The Run-Time Library routine LIB\$GET\_VM returns memory that is quadword aligned. Therefore, you should use LIB\$GET\_VM to allocate the virtual memory for a queue. For instance, to create a COMMON block called QUEUES, use the LINK command with the FILE/OPTIONS qualifier, where FILE.OPT is a linker option file containing the line:

```
PSECT = QUEUES, QUAD
```

A FORTRAN application using processor-shared memory follows:

```
INTEGER*4 FUNCTION INSERT_Q (QENTRY)
COMMON/QUEUES/QHEADER
INTEGER*4 QENTRY(10), QHEADER(2)
INSERT_Q = LIB$INSQHI (QENTRY, QHEADER)
RETURN
END
```

# Access to VMS System Components

## 2.3 Access to VAX Machine Instructions

A BASIC application using processor-shared memory follows:

```
COM (QUEUES) QENTRY%(9), QHEADER%(1)
EXTERNAL INTEGER FUNCTION LIB$INSQHI
IF LIB$INSQHI (QENTRY%() BY REF, QHEADER%() BY REF) AND 1%
    THEN GOTO 1000
.
.
.
1000 REM  INSERTED OK
```

### LIB\$REMQHI

In FORTRAN, the address of the removed queue entry can be passed to another routine as an array using the %VAL built-in function. In the following example, queue entries are ten longwords including the two longword pointers at the beginning of each entry.

```
COMMON/QUEUES/QHEADER
INTEGER*4 QHEADER(2), ISTAT
ISTAT = LIB$REMQHI (QHEADER, ADDR)
IF (ISTAT) THEN
    CALL PROC (%VAL (ADDR)) ! Process removed entry
    GO TO ...
ELSE IF (ISTAT .EQ. %LOC(LIB$_QUEWASEMP)) THEN
    GO TO ... ! Queue was empty
    ELSE IF
        ... ! Secondary interlock failed
END IF
.
.
.
END
SUBROUTINE PROC (QENTRY)
INTEGER*4 QENTRY(10)
.
.
.
RETURN
END
```

### 2.3.4 Character String Routines

The character string routines give a high-level language program access to the corresponding VAX machine instructions. For a complete description of these instructions, see the *VAX Architecture Reference Manual*. For each instruction, the *VAX Architecture Reference Manual* specifies the contents of all the registers after the instruction executes. The corresponding Run-Time Library routines do not make the contents of all the registers available to the calling program.

# Access to VMS System Components

## 2.3 Access to VAX Machine Instructions

**Table 2-6 Character String Routines**

Entry Point	Function
LIB\$LOCC	Locates a character in a string
LIB\$MATCHC	Returns the relative position of a substring
LIB\$SCANC	Scans characters
LIB\$SKPC	Skips characters
LIB\$SPANC	Spans characters
LIB\$MOVC3	Moves characters
LIB\$MOVC5	Moves characters and fills
LIB\$MOVTC	Moves translated characters
LIB\$MOVTUC	Move translated characters until specified character is found

The *VMS RTL String Manipulation (STR\$) Manual* describes STR\$ string manipulation routines.

### Example

This COBOL program uses LIB\$LOCC to return the position of a given letter of the alphabet.

```

IDENTIFICATION DIVISION.
PROGRAM-ID.          LIBLOC.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01  SEARCH-STRING  PIC X(26)
                        VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
01  SEARCH-CHAR    PIC X.
01  IND-POS        PIC 9(9) USAGE IS COMP.
01  DISP-IND       PIC 9(9).

ROUTINE DIVISION.

001-MAIN.
    MOVE SPACE TO SEARCH-CHAR.
    DISPLAY " ".
    DISPLAY "ENTER SEARCH CHARACTER: " WITH NO ADVANCING.
    ACCEPT SEARCH-CHAR.
    CALL "LIB$LOCC"
        USING BY DESCRIPTOR SEARCH-CHAR, SEARCH-STRING
        GIVING IND-POS.
    IF IND-POS = ZERO
        DISPLAY
            "CHAR ENTERED (" SEARCH-CHAR ") NOT A VALID SEARCH CHAR"
        STOP RUN.
    MOVE IND-POS TO DISP-IND.
    DISPLAY
        "SEARCH CHAR (" SEARCH-CHAR ") WAS FOUND IN POSITION "
        DISP-IND.
    GO TO 001-MAIN.

```

# Access to VMS System Components

## 2.3 Access to VAX Machine Instructions

### 2.3.5 Miscellaneous Instruction Routines

Additional routines you may use are listed in Table 2-7.

**Table 2-7 Miscellaneous Instruction Routines**

Entry Point	Function
LIB\$CALLG	Calls a routine using an array argument list
LIB\$CRC	Computes a Cyclic Redundancy Check
LIB\$CRC_TABLE	Constructs a table for a Cyclic Redundancy Check

#### LIB\$CALLG

LIB\$CALLG allows your program access to the CALLG instruction. This instruction calls a routine using an argument list stored as an array in memory, as opposed to the CALLS instruction, in which the argument list is pushed on the stack.

#### LIB\$CRC

LIB\$CRC allows your high-level language program to use the CRC instruction, which calculates the Cyclic Redundancy Check. This instruction is used to check the integrity of a data stream by comparing its state at the sending point and the receiving point. Each character in the data stream is used to generate a value based on a polynomial. The values for each character are then added together. This operation is performed at both ends of the data transmission, and the two result values compared. If the results disagree, then an error occurred during the transmission.

#### LIB\$CRC\_TABLE

LIB\$CRC\_TABLE takes a polynomial as its input and builds the table that LIB\$CRC uses to calculate the CRC. You must specify the polynomial to be used.

For further details, see the *VAX Architecture Reference Manual*.

### 2.4 Processwide Resource Allocation Routines

This section discusses routines that allocate processwide resources to a single VMS process. The processwide resources discussed here are 1) VMS local event flags, and 2) BASIC and FORTRAN logical unit numbers (LUNs). The resource-Allocation Routines are provided so that user routines can use the processwide resources without conflicting with one another.

In general, you must use Run-Time Library resource Allocation Routines when your program needs processwide resources. This allows Run-Time Library routines, DIGITAL-supplied routines, and user routines that you write to perform together within a process.

If your called routine includes a call to any Run-Time Library routine that frees a processwide resource, and that called routine fails to execute normally, the resource will not be freed. Thus, your routine should establish a condition handler that frees the allocated resource before resignaling or unwinding. Chapter 4 describes condition handling.

# Access to VMS System Components

## 2.4 Processwide Resource Allocation Routines

Table 2-8 lists the processwide resource allocation routines.

**Table 2-8 Processwide Resource Allocation Routines**

Entry Point	Function
LIB\$FREE_LUN	Deallocates a specific logical unit number
LIB\$GET_LUN	Allocates next arbitrary logical unit number
LIB\$FREE_EF	Frees a local event flag
LIB\$GET_EF	Allocates a local event flag
LIB\$RESERVE_EF	Reserves a local event flag

### 2.4.1 Allocating Logical Unit Numbers

BASIC and FORTRAN use a *logical unit number* (LUN) to define the file or device a program uses to perform input and output. For a routine to be modular, it does not need to know the unit numbers being used by other routines running at the same time. For this reason, logical units are allocated and deallocated at run time. You can use LIB\$GET\_LUN and LIB\$FREE\_LUN to obtain the next available number. This ensures that your BASIC or FORTRAN routine will not use a logical unit that is already being used by a calling program. Therefore, you should use this routine whenever your program calls or is called by another program which may also allocate LUNs. Logical unit numbers 100 to 119 are available to modular routines through these entry points.

To allocate a LUN, call LIB\$GET\_LUN and use the value returned as the LUN for your I/O statements. If no LUNs are available, an error status is returned and the logical unit is set to -1. When the program unit exits, it should use LIB\$FREE\_LUN to free any LUNs that have been allocated by LIB\$GET\_LUN. If it does not free any LUNs, the available pool of numbers is available for use.

If your called routine contains a call to LIB\$FREE\_LUN to free the LUNs upon exit, and your routine fails to execute normally, the LUNs will not be freed. For this reason, you should make sure to establish a condition handler to call LIB\$FREE\_LUN before resignaling or unwinding. Otherwise, the allocated LUN is lost until the image exits.

### 2.4.2 Allocating Event Flag Numbers

LIB\$GET\_EF and LIB\$FREE\_EF operate in a similar way to LIB\$GET\_LUN and LIB\$FREE\_LUN. They cause local event flags to be allocated and deallocated at run time, so that your routine remains independent of other routines executing in the same process.

Local event flags numbered 32 to 63 are available to your program. These event flags allow routines to communicate and synchronize their operations. If you use a specific event flag in your routine, another routine may attempt to use the same flag, and the flag will no longer function as expected. Therefore, you should call LIB\$GET\_EF to obtain the next arbitrary event flag and LIB\$FREE\_EF to return it to the storage pool. You can obtain a specific event flag number by calling LIB\$RESERVE\_EF. This routine takes as its argument the event flag number to be allocated.

# Access to VMS System Components

## 2.5 Performance Measurement Routines

### 2.5 Performance Measurement Routines

The Run-Time Library timing facility consists of four routines to store count and timing information, display the requested information, and deallocate the storage. Table 2-9 lists these routines and their functions.

**Table 2-9 Performance Measurement Routines**

Entry Point	Function
LIB\$INIT_TIMER	Stores the values of the specified times and counts in units of static or heap storage, depending on the value of the routine's argument
LIB\$SHOW_TIMER	Gets and formats for output the specified times and counts that are accumulated since the last call to LIB\$INIT_TIMER
LIB\$STAT_TIMER	Gets one of the times and counts since the last call to LIB\$INIT_TIMER and returns it as an unsigned quadword or longword
LIB\$FREE_TIMER	Frees the storage allocated by LIB\$INIT_TIMER

Using these routines, you can access the following statistics:

- Elapsed time
- CPU time
- Buffered I/O count
- Direct I/O count
- Page faults

LIB\$SHOW\_TIMER and LIB\$STAT\_TIMER are relatively simple tools for testing the performance of a new application. To obtain more detailed information, use the VMS system services SYS\$GETTIM (Get Time) and SYS\$GETJPI (Get Job/Process Information).

The simplest way to use the Run-Time Library routines is to call LIB\$INIT\_TIMER with no arguments at the beginning of the portion of code to be monitored. This will cause the statistics to be placed in OWN storage. To get the statistics from OWN storage, call LIB\$SHOW\_TIMER (with no arguments) at the end of the portion of code to be monitored.

If you want a particular statistic, you must include a **code** argument with a call to LIB\$SHOW\_TIMER or LIB\$STAT\_TIMER, as shown in Table 2-10. LIB\$SHOW\_TIMER returns the specified statistics in formatted form and sends them to SYS\$OUTPUT. On each call, LIB\$STAT\_TIMER returns one statistic to the calling program as an unsigned longword or quadword value.

# Access to VMS System Components

## 2.5 Performance Measurement Routines

**Table 2-10 The Code Argument in LIB\$SHOW\_TIMER and LIB\$STAT\_TIMER**

Value of	Meaning	Format	Format
1	Elapsed real time	hhhh:mm:ss.cc	Quadword, in system time format
2	Elapsed CPU time	hhhh:mm:ss.cc	Longword, in 10-millisecond increments
3	Count of buffered I/O operations	nnnn	Longword
4	Count of direct I/O operations	nnnn	Longword
5	Count of page faults	nnnn	Longword

When you call LIB\$INIT\_TIMER, you must use the optional **handler** argument only if you want to keep several sets of statistics simultaneously. This argument points to a block in heap storage where the statistics are to be stored. You only need to call LIB\$FREE\_TIMER if you have specified **handler** in LIB\$INIT\_TIMER and you want to deallocate all heap storage resources. In most cases, the implicit deallocation when the image exits will be sufficient.

LIB\$STAT\_TIMER returns only one of the five statistics for each call, and returns that statistic in the form of an unsigned quadword or longword. LIB\$SHOW\_TIMER returns the virtual address of the stored information, which BASIC cannot directly access. Therefore, a BASIC program must call LIB\$STAT\_TIMER and format the returned statistics, as the following example demonstrates.

### Example

The following BASIC example uses the Run-Time Library performance analysis routines to obtain timing statistics. It then calls the \$ASCTIM system service to translate the 64-bit binary value returned by LIB\$STAT\_TIMER into an ASCII text string.

```

100  EXTERNAL INTEGER FUNCTION LIB$INIT_TIMER
      EXTERNAL INTEGER FUNCTION LIB$STAT_TIMER
      EXTERNAL INTEGER FUNCTION LIB$FREE_TIMER
      EXTERNAL INTEGER CONSTANT SS$_NORMAL

200  DECLARE LONG COND_VALUE, RANDOM_SLEEP
      DECLARE LONG CODE, HANDLE
      DECLARE STRING TIME_BUFFER
      HANDLE = 0
      TIME_BUFFER = SPACE$(50%)

300  MAP (TIMER) LONG ELAPSED_TIME, FILL
      MAP (TIMER) LONG CPU_TIME
      MAP (TIMER) LONG BUFIO
      MAP (TIMER) LONG DIRIO
      MAP (TIMER) LONG PAGE_FAULTS

```



# Access to VMS System Components

## 2.5 Performance Measurement Routines

```
400 PRINT "This program returns information about:"
PRINT "Elapsed time (1)"
PRINT "CPU time (2)"
PRINT "Buffered I/O (3)"
PRINT "Direct I/O (4)"
PRINT "Page faults (5)"
PRINT "Enter zero to exit program"
PRINT "Enter a number from one to"
PRINT "five for performance information"
INPUT "One, two, three, four, or five"; CODE
PRINT

450 GOTO 32766 IF CODE = 0

500 COND_VALUE = LIB$INIT_TIMER( HANDLE )

550 IF (COND_VALUE <> SS$_NORMAL) THEN PRINT @
    "Error in initialization"
    GOTO 32767

650 A = 0 !
FOR I = 1 to 100000 ! This code merely uses some CPU time
A = A + 1 !
NEXT I !

700 COND_VALUE = LIB$STAT_TIMER( CODE, ELAPSED_TIME, HANDLE )

750 IF (COND_VALUE <> SS$_NORMAL) THEN PRINT @
    "Error in statistics routine"
    GOTO 32767

800 GOTO 810 IF CODE <> 1%
CALL SYS$ASCTIM ( , TIME_BUFFER, ELAPSED_TIME, 1% BY VALUE)
PRINT "Elapsed time: "; TIME_BUFFER

810 PRINT "CPU time in seconds: "; .01 * CPU_TIME IF CODE = 2%
PRINT "Buffered I/O: "; BUFIO IF CODE = 3%
PRINT "Direct I/O: "; DIRIO IF CODE = 4%
PRINT "Page faults: "; PAGE_FAULTS IF CODE = 5%
PRINT

900 GOTO 400

32765 COND_VALUE = LIB$FREE_TIMER( HANDLE )
32766 IF (COND_VALUE <> SS$_NORMAL) THEN PRINT @
    "Error in LIB$FREE_TIMER"
    GOTO 32767

32767 END
```

---

## 2.6 Output Formatting Control Routines

Table 2-11 lists the Run-Time Library routines that customize output.

**Table 2–11 Routines for Customizing Output**

Entry Point	Function
LIB\$CURRENCY	Defines the default currency symbol for process
LIB\$DIGIT_SEP	Defines the default digit separator for process
LIB\$LP_LINES	Defines the process default size for a printed page
LIB\$RADIX_POINT	Defines the process default radix point character

LIB\$CURRENCY, LIB\$DIGIT\_SEP, LIB\$LP\_LINES, and LIB\$RADIX\_POINT allow you to customize output. Using them, you can define the logical names SYS\$CURRENCY, SYS\$DIGIT\_SEP, SYS\$LP\_LINES, and SYS\$RADIX\_POINT to specify your own currency symbol, digit separator, radix point, or number of lines per printed page. Each of these routines works by attempting to translate the associated logical name as a process, group, or system logical name. If you have redefined a logical name for a specific local application, then the translation succeeds, and the routine returns the value that corresponds to the option you have chosen. If the translation fails, the routine returns a default value provided by the Run-Time Library, as follows:

```
$  SYS$CURRENCY
,  SYS$DIGIT_SEP
.  SYS$RADIX_POINT
66 SYS$LP_LINES
```

For example, if you want to use the British pound sign as the currency symbol within your process, but you want to leave the dollar sign as the system's default, define SYS\$CURRENCY to be in your process logical name table. After this, any call to LIB\$CURRENCY within your process returns "£", while any call outside your process returns "\$".

You can use LIB\$LP\_LINES to monitor the current default length of the line printer page. You can also supply your own default length for the current process. United States standard paper stock permits 66 lines on each physical page.

If you are writing programs for a utility that formats a listing file to be printed on a line printer, you can use LIB\$LP\_LINES to make your utility independent of the default page length. Your program can use LIB\$LP\_LINES to obtain the current length of the page. It can then calculate the number of lines of text per page by subtracting the lines used for margins and headings.

The following is one suggested format:

- 1** Three lines for the top margin
- 2** Three lines for the bottom margin
- 3** Three lines for listing heading information, consisting of the following:
  - a.** Language-processor identification line
  - b.** Source-program identification line
  - c.** One blank line

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

### 2.7 Miscellaneous Interface Routines

There are several other Run-Time Library routines that permit high-level access to components of VMS. Table 2-12 lists these routines and their functions. The sections that follow give further details about some of these routines.

**Table 2-12 Miscellaneous Interface Routines**

Entry Point	Function
LIB\$AST_IN_PROG	Indicates whether an asynchronous system trap is in progress
LIB\$ASN_WTH_MBX	Assigns an I/O channel and associates it with a mailbox
LIB\$CREATE_DIR	Creates a directory or subdirectory
LIB\$FIND_IMAGE_SYMBOL	Reads a global symbol from the shareable image file and dynamically activates a shareable image into the PO address space of a process
LIB\$ADDX	Performs addition on signed two's complement integers of arbitrary length (multiple-precision addition)
LIB\$SUBX	Performs subtraction on signed two's complement integers of arbitrary length (multiple-precision subtraction)
LIB\$FILE_SCAN	Finds file names given RMS FAB
LIB\$FILE_SCAN_END	End of file scan
LIB\$FIND_FILE	Finds file names given string
LIB\$FIND_FILE_END	End of find file
LIB\$INSERT_TREE	Inserts an element in a binary tree
LIB\$LOOKUP_TREE	Finds an element in a binary tree
LIB\$TRAVERSE_TREE	Traverses a binary tree
LIB\$GET_COMMON	Gets a record from the process's COMMON storage area
LIB\$PUT_COMMON	Puts a record to the process's COMMON storage area

#### 2.7.1 Indicating Asynchronous System Trap in Progress

An asynchronous system trap (AST) is a VMS mechanism for providing a software interrupt when an external event occurs, such as the user typing CTRL/C. When an external event occurs, VMS interrupts the execution of the current process and calls a routine that you supply. While that routine is active, the AST is said to be in progress, and the process is said to be executing at AST level. When your AST routine returns control to the original process, the AST is no longer active and execution continues where it left off.

LIB\$AST\_IN\_PROG indicates to the calling program whether an AST is currently in progress. Your program can call LIB\$AST\_IN\_PROG to determine whether it is executing at AST level, and then take appropriate action. This routine is useful if you are writing AST-reentrant code.

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

### 2.7.2 Assigning an I/O Channel Along with a Mailbox

A mailbox is a virtual device used for communication between processes. A channel is the communication path that a process uses to perform I/O operations to a particular device. LIB\$ASN\_WTH\_MBX assigns a channel to a device and associates a mailbox with the device.

Normally, a process calls the \$CREMBX system service to create a mailbox and assign a channel and logical name to it. In the case of a temporary mailbox, this service places the logical name corresponding to the mailbox in the group logical name table. This implies that any process running in the same group and using the same logical name uses the same mailbox.

Sometimes it is not desirable to have more than one process use the same mailbox. For example, when a program connects explicitly with another process across a network, the program uses a mailbox to obtain the data confirming the connection and to store the asynchronous messages from the other process. If that mailbox is shared with other processes in the same group, there is no way to determine which messages are intended for which processes; the processes read each other's messages, and the original program does not receive the correct information from the cooperating process across the network link.

LIB\$ASN\_WTH\_MBX avoids this situation by associating the physical mailbox name with the channel assigned to the device. To create a temporary mailbox for itself and other processes cooperating with it, your program calls LIB\$ASN\_WTH\_MBX. The Run-Time Library routine assigns the channel and creates the temporary mailbox by using the system services \$GETCHN, \$ASSIGN, and \$CREMBX. Instead of a logical name, the mailbox is identified by a physical device name of the form *MBcu*. The elements which make up this device name are as follows:

MB indicates that the device is a mailbox  
c is the controller  
u is the unit number

The routine returns this device name to the calling program, which then must pass the mailbox channel to the other programs with which it cooperates. In this way, the cooperating processes access the mailbox by its physical name, instead of by its group-wide logical name.

The calling program passes the routine a device name, which specifies the device to which the channel is to be assigned. For this argument (called **dev-nam**), you may use a logical name. If you do so, the routine attempts one level of logical name translation.

The privilege restrictions and process quotas required for using this routine are those required by the \$GETCHN, \$CREMBX, and \$ASSIGN system services.

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

### 2.7.3 Create a Directory or Subdirectory

LIB\$CREATE\_DIR creates a directory or a subdirectory. The calling program must specify the directory specification in standard RMS format. This directory specification may also contain a disk specification.

In addition to the required directory specification argument, LIB\$CREATE\_DIR takes the following five optional arguments:

- The User Identification Code (UIC) of the owner of the created directory or subdirectory
- The protection enable mask
- The protection value mask
- The maximum number of versions allowed for files created in this directory or subdirectory
- The relative volume number within the volume set on which the directory or subdirectory is created

See the reference section of this manual for a complete description of LIB\$CREATE\_DIR.

### 2.7.4 File Searching Routines

The Run-Time Library provides two routines that your program can call to search for a file and two routines that your program can call to end a search sequence.

- When you call LIB\$FILE\_SCAN with a wildcard file specification and an action routine, the routine calls the action routine for each file or error, or both, found in the wildcard sequence. LIB\$FILE\_SCAN allows the search sequence to continue even though certain errors are present.
- When you call LIB\$FIND\_FILE with a wildcard file specification, it finds the next file specification that matches the wildcard specification.

In addition to the wildcard file specification, which is a required argument, LIB\$FIND\_FILE takes the following four optional arguments:

- The default specification.
- The related specification.
- The RMS secondary status value from a failing RMS operation.
- A longword containing two flag bits. If bit 1 is set, LIB\$FIND\_FILE performs temporary defaulting for multiple input files and the related specification argument is ignored. See the reference section of this manual for a complete description of LIB\$FIND\_FILE in template format.

LIB\$FIND\_FILE\_END is called once after each call to LIB\$FIND\_FILE in interactive use. LIB\$FIND\_FILE\_END prevents the temporary default values retained by the previous call to LIB\$FIND\_FILE from affecting the next file specification.

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

LIB\$FILE\_SCAN uses an optional context argument to perform temporary defaulting for multiple input files. For example, a command such as the following would specify A, B, and C in successive calls, retaining context, so that portions of one file specification would affect the next file specification:

```
$ COPY [smith]A,B,C *
```

LIB\$FILE\_SCAN\_END is called once after each sequence of calls to LIB\$FILE\_SCAN. LIB\$FILE\_SCAN\_END performs a parse of the null string to deallocate saved RMS context and to prevent the temporary default values retained by the previous call to LIB\$FILE\_SCAN from affecting the next file specification. For instance, in the previous example, LIB\$FILE\_SCAN\_END should be called after the C file specification is parsed, so that specifications from the \$COPY files do not affect file specifications in subsequent commands.

The following BLISS example illustrates the use of LIB\$FIND\_FILE. It prompts for a file specification and default specification. The default specification indicates the default information for the file for which you are searching. Once the routine has searched for one file, the resulting file specification determines the related file specification and the default file specification for the next search. LIB\$FIND\_FILE\_END is called at the end of this BLISS program to deallocate the virtual memory used by LIB\$FIND\_FILE.

```
%TITLE 'FILE_EXAMPLE1 - Sample program using LIB$FIND_FILE'
MODULE FILE_EXAMPLE1(          ! Sample program using LIB$FIND_FILE
    IDENT = '1-001',
    MAIN = EXAMPLE_START
) =

BEGIN

%SBTTL 'Declarations'
!+
! SWITCHES:
!-

SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL, NONEXTERNAL = WORD_RELATIVE);

!+
! TABLE OF CONTENTS:
!-

FORWARD ROUTINE
    EXAMPLE_START;                ! Main program

!+
! INCLUDE FILES:
!-

LIBRARY 'SYS$LIBRARY:STARLET.L32';    ! System symbols

!+
! Define facility-specific messages from shared system messages.
!-
$SHR_MSGDEF(CLI,3,LOCAL,
            (PARSEFAIL,WARNING));

!+
! EXTERNAL REFERENCES:
!-
```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```
EXTERNAL ROUTINE
  LIB$GET_INPUT,           ! Read from SYS$INPUT
  LIB$FIND_FILE,          ! Wildcard scanning routine
  LIB$FIND_FILE_END,      ! End find file
  LIB$PUT_OUTPUT,        ! Write to SYS$OUTPUT
  STR$COPY_DX;           ! String copier

LITERAL
  TRUE = 1,              ! Success
  FALSE = 0;            ! Failure

%SBTTL 'EXAMPLE_START - Sample program main routine';
ROUTINE EXAMPLE_START =
BEGIN
!+
! This program reads a file specification and default file
! specification from SYS$INPUT. It then prints all the files that
! match that specification and prompts for another file specification.
! After the first file specification no default specification is requested,
! and the previous resulting file specification becomes the related
! file specification.
!-
LOCAL
  LINEDESC : $BLOCK[DSC$C_S_BLN], ! String desc. for input line
  RESULT_DESC : $BLOCK[DSC$C_S_BLN], ! String desc. for result file
  CONTEXT, ! LIB$FIND_FILE context pointer
  DEFAULT_DESC : $BLOCK[DSC$C_S_BLN], ! String desc. for default spec
  RELATED_DESC : $BLOCK[DSC$C_S_BLN], ! String desc. for related spec
  HAVE_DEFAULT,
  STATUS;

!+
! Make all string descriptors dynamic.
!-
CH$FILL(0,DSC$C_S_BLN,LINEDESC);
LINEDESC[DSC$B_CLASS] = DSC$K_CLASS_D;
CH$MOVE(DSC$C_S_BLN,LINEDESC,RESULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,DEFAULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,RELATED_DESC);
HAVE_DEFAULT = FALSE;
CONTEXT = 0;
!+
! Read file specification, default file specification, and
! related file specification.
!-
```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```

WHILE (STATUS = LIB$GET_INPUT(LINEDESC,
                             $DESCRIPTOR('FILE SPECIFICATION: '))) NEQ RMS$_EOF
DO BEGIN
  IF NOT .STATUS
    THEN SIGNAL_STOP(.STATUS);
  !+
  ! If default file specification was not obtained, do so now.
  !-
  IF NOT .HAVE_DEFAULT
  THEN BEGIN
    STATUS = LIB$GET_INPUT(DEFAULT_DESC,
                          $DESCRIPTOR('DEFAULT FILE SPECIFICATION: '));
    IF NOT .STATUS
      THEN SIGNAL_STOP(.STATUS);
    HAVE_DEFAULT = TRUE;
  END;
  !+
  ! CALL LIB$FIND_FILE until RMS$_NMF (no more files) is returned.
  ! If an error other than RMS$_NMF is returned, it is signaled.
  ! Print out the file specification if the call is successful.
  !-
  WHILE (STATUS = LIB$FIND_FILE(LINEDESC,RESULT_DESC,CONTEXT,
                               DEFAULT_DESC,RELATED_DESC)) NEQ RMS$_NMF
  DO IF NOT .STATUS
    THEN SIGNAL(CLI$_PARSEFAIL,1,RESULT_DESC,.STATUS)
    ELSE LIB$PUT_OUTPUT(RESULT_DESC);
  !+
  ! Make this resultant file specification the related file
  ! specification for next file.
  !-
  STR$COPY_DX(RELATED_DESC,LINEDESC);
  END;
                                     ! End of loop
                                     ! reading file specification

!+
! Call LIB$FIND_FILE_END to deallocate the virtual memory used by LIB$FIND_FILE.
! Note that we do this outside of the loop. Since the MULTIPLE bit of the
! optional user flags argument to LIB$FIND_FILE wasn't used, it is not
! necessary to call LIB$FIND_FILE_END after each call to LIB$FIND_FILE.
! (The MULTIPLE bit would have caused temporary defaulting for multiple input
! files.)
!-
STATUS = LIB$FIND_FILE_END (CONTEXT);

IF NOT .STATUS
  THEN SIGNAL_STOP (.STATUS);

RETURN TRUE
END;
                                     ! End of main program
END
                                     ! End of module

ELUDOM

```

This BLISS example illustrates LIB\$FILE\_SCAN and LIB\$FILE\_SCAN\_END.

```

%TITLE 'FILE_EXAMPLE2 - Sample program using LIB$FILE_SCAN'
MODULE FILE_EXAMPLE1(
  IDENT = '1-001',
  MAIN = EXAMPLE_START
) =
BEGIN

```



# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```
%SBTTL 'Declarations'
!+
! SWITCHES:
!-

SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL,
    NONEXTERNAL = WORD_RELATIVE);

!+
! TABLE OF CONTENTS:
!-

FORWARD ROUTINE
    EXAMPLE_START,          ! Main program
    SUCCESS_RTN,           ! Success action routine
    ERROR_RTN;             ! Error action routine

!+
! INCLUDE FILES:
!-

LIBRARY 'SYS$LIBRARY:STARLET.L32';      ! System symbols

!+
! Define VMS block structures (BLOCK[,BYTE]).
!-
STRUCTURE
    BBLOCK [0, P, S, E; N] =
        [N]
        (BBLOCK + 0) <P, S, E>;

!+
! EXTERNAL REFERENCES:
!-

EXTERNAL ROUTINE
    LIB$GET_INPUT,          ! Read from SYS$INPUT
    LIB$FILE_SCAN,         ! Wildcard scanning routine
    LIB$FILE_SCAN_END,     ! End of file scan
    LIB$PUT_OUTPUT;        ! Write to SYS$OUTPUT

%SBTTL 'EXAMPLE_START - Sample program main routine';
ROUTINE EXAMPLE_START =
BEGIN
!+
! This program reads the file specification, default file specification,
! and related file specification from SYS$INPUT and then displays on
! SYS$OUTPUT all files which match the specification.
!-
LOCAL
    RESULT_BUFFER : VECTOR[NAM$C_MAXRSS,BYTE], !Buffer for resultant
        ! name string
    EXPAND_BUFFER : VECTOR[NAM$C_MAXRSS,BYTE], !Buffer for expanded
        ! name string
    LINEDESC : BBLOCK[DSC$C_S_BLN],           !String descriptor
        ! for input line
    RESULT_DESC : BBLOCK[DSC$C_S_BLN],        !String descriptor
        ! for result file
    DEFAULT_DESC : BBLOCK[DSC$C_S_BLN],       !String descriptor
        ! for default specification
    RELATED_DESC : BBLOCK[DSC$C_S_BLN],       !String descriptor
        ! for related specification
    IFAB : $FAB_DECL,                         !FAB for file_scan
    INAM : $NAM_DECL,                         ! and a NAM block
    RELNAM : $NAM_DECL,                       ! and a related NAM block
    STATUS;
```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```

!+
! Make all descriptors dynamic.
!-
CH$FILL(0,DSC$C_S_BLN,LINEDESC);
LINEDESC[DSC$B_CLASS] = DSC$K_CLASS_D;
CH$MOVE(DSC$C_S_BLN,LINEDESC,RESULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,DEFAULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,RELATED_DESC);
!+
! Read file specification, default file specification, and related
! file specification
!-
STATUS = LIB$GET_INPUT(LINEDESC,
    $DESCRIPTOR('File specification: '));
IF NOT .STATUS
    THEN SIGNAL_STOP(.STATUS);
STATUS = LIB$GET_INPUT(DEFAULT_DESC,
    $DESCRIPTOR('Default file specification: '));
IF NOT .STATUS
    THEN SIGNAL_STOP(.STATUS);
STATUS = LIB$GET_INPUT(RELATED_DESC,
    $DESCRIPTOR('Related file specification: '));
IF NOT .STATUS
    THEN SIGNAL_STOP(.STATUS);
!+
! Initialize the FAB, NAM, and related NAM blocks.
!-
$FAB_INIT(FAB=IFAB,
    FNS=.LINEDESC[DSC$W_LENGTH],
    FNA=.LINEDESC[DSC$A_POINTER],
    DNS=.DEFAULT_DESC[DSC$W_LENGTH],
    DNA=.DEFAULT_DESC[DSC$A_POINTER],
    NAM=INAM);

$NAM_INIT(NAM=INAM,
    RSS=NAM$C_MAXRSS,
    RSA=RESULT_BUFFER,
    ESS=NAM$C_MAXRSS,
    ESA=EXPAND_BUFFER,
    RLF=RELNAM);

$NAM_INIT(NAM=RELNAM);
RELNAM[NAM$B_RSL] = .RELATED_DESC[DSC$W_LENGTH];
RELNAM[NAM$L_RSA] = .RELATED_DESC[DSC$A_POINTER];
!+
! Call LIB$FILE_SCAN. Note that errors need not be checked
! here because LIB$FILE_SCAN calls error_rtn for all errors.
!-
LIB$FILE_SCAN(IFAB,SUCCESS_RTN,ERROR_RTN);

!+
! Call LIB$FILE_SCAN_END to deallocate virtual memory used for
! file scan structures.
!-
STATUS = LIB$FILE_SCAN_END (IFAB);

IF NOT .STATUS
    THEN SIGNAL_STOP (.STATUS);

RETURN 1
END;
! End of main program

```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```
ROUTINE SUCCESS_RTN (IFAB : REF BBLOCK) =
BEGIN
!+
! This routine is called by LIB$FILE_SCAN for each file that it
! successfully finds in the search sequence.
!
! Inputs:
!
!     IFAB      Address of a fab
!
! Outputs:
!
!     file specification printed on SYS$OUTPUT
!-
LOCAL
    DESC : BBLOCK[DSC$C_S_BLN];      ! A local string descriptor
BIND
    INAM = .IFAB[FAB$L_NAM] : BBLOCK;    ! Find NAM block
                                           ! from pointer in FAB
CH$FILL(0,DSC$C_S_BLN,DESC);           ! Make static
                                           ! string descriptor
DESC[DSC$W_LENGTH] = .INAM[NAM$B_RSL];  ! Get string length
                                           ! from NAM block
DESC[DSC$A_POINTER] = .INAM[NAM$L_RSA];  ! Get pointer to the string
RETURN LIB$PUT_OUTPUT(DESC)             ! Print name on SYS$OUTPUT
                                           ! and return
END;

ROUTINE ERROR_RTN (IFAB : REF BBLOCK) =
BEGIN
!+
! This routine is called by LIB$FILE_SCAN for each file specification that
! produces an error.
!
! Inputs:
!
!     ifab      Address of a fab
!
! Outputs:
!
!     Error message is signaled
!-
LOCAL
    DESC : BBLOCK[DSC$C_S_BLN];      ! A local string descriptor
BIND
    INAM = .IFAB[FAB$L_NAM] : BBLOCK;    ! Get NAM block pointer
                                           ! from FAB
CH$FILL(0,DSC$C_S_BLN,DESC);           ! Create static
                                           ! string descriptor
DESC[DSC$W_LENGTH] = .INAM[NAM$B_RSL];
DESC[DSC$A_POINTER] = .INAM[NAM$L_RSA];
!+
! Signal the error using the shared message PARSEFAIL
! and the CLI facility code. The second part of the SIGNAL
! is the RMS STS and STV error codes.
!-
RETURN SIGNAL((SHR$_PARSEFAIL+3*16),1,DESC,
              .IFAB[FAB$L_STS],.IFAB[FAB$L_STV])

END;
END
                                ! End of module
ELUDOM
```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

### 2.7.5 Insert Entry in a Balanced Binary Tree

Three routines allow you to manipulate the contents of a balanced binary tree:

- LIB\$INSERT\_TREE adds an entry to a balanced binary tree.
- LIB\$LOOKUP\_TREE looks up an entry in a balanced binary tree.
- LIB\$TRAVERSE\_TREE calls an action routine for each node in the tree.

#### Example

The following BLISS example illustrates all three of these routines. The program prompts for input from SYS\$INPUT and stores each data line as an entry in a binary tree. When the user enters end-of-file character (CTRL/Z), the tree will be printed in sorted order. The program includes three subroutines:

- The first subroutine allocates virtual memory for a node.
- The second subroutine routine compares a key with a node.
- The third subroutine is called during the tree traversal. It prints out the left and right subtree pointers, the current node balance, and the name of the node.

```
%TITLE 'TREE_EXAMPLE - Sample program using binary tree routines'
MODULE TREE_EXAMPLE(                                     ! Sample program using trees
    IDENT = '1-001',
    MAIN = TREE_START
) =

BEGIN

%SBTTL 'Declarations'
!+
! SWITCHES:
!-
SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL, NONEXTERNAL = WORD_RELATIVE);

!+
! LINKAGES:
!
!     NONE
!
! TABLE OF CONTENTS:
!-

FORWARD ROUTINE
    TREE_START,                                     ! Main program
    ALLOC_NODE,                                     ! Allocate memory for a node
    COMPARE_NODE,                                   ! Compare two nodes
    PRINT_NODE;                                     ! Print a node (action routine
                                                    ! for LIB$TRAVERSE_TREE)

!+
! INCLUDE FILES:
!-

LIBRARY 'SYS$LIBRARY:STARLET.L32';                 ! System symbols
```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```
!+
! Define VMS block structures (BLOCK[,BYTE]).
!-
STRUCTURE
    BBLOCK [0, P, S, E; N] =
        [N]
        (BBLOCK + 0) <P, S, E>;
!+
! MACROS:
!-
MACRO
    NODE$L_LEFT = 0,0,32,0%,      ! Left subtree pointer in node
    NODE$L_RIGHT = 4,0,32,0%,     ! Right subtree pointer
    NODE$W_BAL = 8,0,16,0%,       ! Balance this node
    NODE$B_NAMLANG = 10,0,8,0%,   ! Length of name in this node
    NODE$T_NAME = 11,0,0,0%;      ! Start of name (variable length)
LITERAL
    NODE$C_LENGTH = 11;           ! Length of fixed part of node
!+
! EXTERNAL REFERENCES:
!-
EXTERNAL ROUTINE
    LIB$GET_INPUT,                ! Read from SYS$INPUT
    LIB$GET_VM,                   ! Allocate virtual memory
    LIB$INSERT_TREE,              ! Insert into binary tree
    LIB$LOOKUP_TREE,              ! Lookup in binary tree
    LIB$PUT_OUTPUT,               ! Write to SYS$OUTPUT
    LIB$TRAVERSE_TREE,            ! Traverse a binary tree
    STR$UPCASE,                   ! Convert string to all uppercase
    SYS$FAO;                       ! Formatted ASCII output routine
%SBTTL 'TREE_START - Sample program main routine';
ROUTINE TREE_START =
BEGIN
!+
! This program reads from SYS$INPUT and stores each data line
! as an entry in a binary tree. When end-of-file character (CTRL/Z)
! is entered, the tree will be printed in sorted order.
!-
LOCAL
    NODE : REF BBLOCK,            ! Address of allocated node
    TREEHEAD,                     ! List head of binary tree
    LINEDESC : BBLOCK[DSC$C_S_BLN], ! String descriptor for input line
    STATUS;
```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```

TREEHEAD = 0;                                ! Zero binary tree head
CH$FILL(0,DSC$C_S_BLN,LINEDESC);            ! Make a dynamic descriptor
LINEDESC[DSC$B_CLASS] = DSC$K_CLASS_D;      ! ...
!+
! Read input lines until end of file seen.
!-
WHILE (STATUS = LIB$GET_INPUT(LINEDESC,      ! Read input line
                             $DESCRIPTOR('Text: '))) ! with this prompt
    NEQ RMS$_EOF
DO IF NOT .STATUS                            ! Report any errors found
    THEN SIGNAL(.STATUS)
    ELSE BEGIN
        STR$UPCASE(LINEDESC,LINEDESC);      ! Convert string
                                           ! to uppercase
        IF NOT (STATUS = LIB$INSERT_TREE(
            TREEHEAD,                        ! Insert good data into the tree
            LINEDESC,                        ! Data to insert
            %REF(1),                         ! Insert duplicate entries
            COMPARE_NODE,                   ! Addr. of compare routine
            ALLOC_NODE,                     ! Addr. of node allocation routine
            NODE,                            ! Return addr. of
            0))                              ! allocated node here
        THEN SIGNAL(.STATUS);
    END;
!+
! End of file character encountered. Print the whole tree and exit.
!-
IF NOT (STATUS = LIB$TRAVERSE_TREE(
    TREEHEAD,                                ! Listhead of tree
    PRINT_NODE,                              ! Action routine to print a node
    0))
    THEN SIGNAL(.STATUS);

RETURN SS$_NORMAL
END;                                          ! End of routine tree_start

ROUTINE ALLOC_NODE (KEYDESC,RETDESC,CONTEXT) =
BEGIN
!+
! This routine allocates virtual memory for a node.
!
! INPUTS:
!
!     KEYDESC      Address of string descriptor for key
!                  (this is the linedesc argument passed
!                  to LIB$INSERT_TREE)
!     RETDESC      Address of location to return address of
!                  allocated memory
!     CONTEXT      Address of user context argument passed
!                  to LIB$INSERT_TREE (not used in this
!                  example)
!
! OUTPUTS:
!
!     Memory address returned in longword pointed to by retdesc
!-
MAP
    KEYDESC : REF BBLOCK,
    RETDESC : REF VECTOR[,LONG];

LOCAL
    NODE : REF BBLOCK,
    STATUS;

```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```
STATUS = LIB$GET_VM(%REF(NODE$C_LENGTH+.KEYDESC[DSC$W_LENGTH]), NODE);
IF NOT .STATUS
  THEN RETURN .STATUS
  ELSE BEGIN
    NODE[NODE$B_NAMLNG] = .KEYDESC[DSC$W_LENGTH]; ! Set name length
    CH$MOVE(.KEYDESC[DSC$W_LENGTH], ! Copy in the name
            .KEYDESC[DSC$A_POINTER],
            NODE[NODE$T_NAME]);
    RETDESC[0] = .NODE; ! Return address to caller
    END;
RETURN .STATUS

END;

ROUTINE COMPARE_NODE (KEYDESC, NODE, CONTEXT) =
BEGIN
!+
! This routine compares a key with a node.
!
! INPUTS:
!
! KEYDESC Address of string descriptor for new key
! (This is the linedesc argument passed to
! LIB$INSERT_TREE)
! NODE Address of current node
! CONTEXT User context data (Not used in this example)
!-
MAP
  KEYDESC : REF BBLOCK,
  NODE : REF BBLOCK;

RETURN CH$COMPARE(.KEYDESC[DSC$W_LENGTH], ! Compare key with
                  ! current node
                  .KEYDESC[DSC$A_POINTER],
                  .NODE[NODE$B_NAMLNG],
                  NODE[NODE$T_NAME])

END;

ROUTINE PRINT_NODE (NODE, CONTEXT) =
BEGIN
!+
! This routine is called during the tree traversal. It
! prints out the left and right subtree pointers, the
! current node balance, and the name of the node.
!-
MAP
  NODE : REF BBLOCK;
```

# Access to VMS System Components

## 2.7 Miscellaneous Interface Routines

```
LOCAL
  OUTBUF : BBLOCK[512],           ! FAO output buffer
  OUTDESC : BBLOCK[DSC$C_S_BLN], ! Output buffer descriptor
  STATUS;
CH$FILL(0,DSC$C_S_BLN,OUTDESC); ! Zero descriptor
OUTDESC[DSC$W_LENGTH] = 512;
OUTDESC[DSC$A_POINTER] = OUTBUF;
IF NOT (STATUS = SYS$FAO($DESCRIPTOR('!XL !XL !XL !XW !AC'),
  OUTDESC,OUTDESC,
  .NODE,.NODE[NODE$L_LEFT],
  .NODE[NODE$L_RIGHT],
  .NODE[NODE$W_BAL],
  NODE[NODE$B_NAMNG]))
THEN SIGNAL(.STATUS)
ELSE BEGIN
  STATUS = LIB$PUT_OUTPUT(OUTDESC); ! Output the line
  IF NOT .STATUS
  THEN SIGNAL(.STATUS);
END;

RETURN SS$_NORMAL

END;
END                                     ! End of module TREE_EXAMPLE
ELUDOM
```

### 2.7.6 Common I/O Routines

LIB\$PUT\_COMMON allows a program to copy a string into the process's common storage area. This area remains defined during multiple image activations. LIB\$GET\_COMMON allows a program to copy a string from the common area into a destination string. The programs reading and writing the data in the common area must agree upon its amount and format. The maximum length of the destination string is defined as follows:

[min(256, the length of the data in the common storage area) - 4]

This maximum length is normally 252.

In BASIC and FORTRAN, you can use these routines to allow a USEROPEN routine to pass information back to the routine that called it. A USEROPEN routine cannot write arguments. However, it can call LIB\$PUT\_COMMON to put information into the common area. The calling program can then use LIB\$GET\_COMMON to retrieve it.

You can also use these routines to pass information between images run successively, such as chained images run by LIB\$RUN\_PROGRAM.





# 3

## Date/Time Manipulation

This chapter describes the routines provided by the Run-Time Library to perform date/time manipulation. These date/time routines return information about a date or time, perform various arithmetic functions on dates and times, and format dates and times in formats other than the standard VMS format.

The following table lists all the LIB\$ routines that perform date/time manipulation.

**Table 3–1 Date/Time Formatting Routines**

<b>Routine</b>	<b>Function</b>
LIB\$ADD_TIMES	Adds two quadword times
LIB\$CONVERT_DATE_STRING	Converts an input date/time string to a VMS internal time
LIB\$CVT_FROM_INTERNAL_TIME	Converts a VMS standard internal binary time value to an external integer value
LIB\$CVTF_FROM_INTERNAL_TIME	Converts a VMS standard internal binary time to an external F-floating point value
LIB\$CVT_TO_INTERNAL_TIME	Converts an external integer time value to a VMS standard internal binary time value
LIB\$CVTF_TO_INTERNAL_TIME	Converts an F-floating point time value to an internal binary time value
LIB\$CVT_VECTIM	Converts a seven-word array to a VMS standard format internal time
LIB\$DATE_TIME	Returns the system date and time in the semantics of the user's string
LIB\$DAY	Returns the number of days since November 17, 1858
LIB\$DAY_OF_WEEK	Returns the numeric day of the week for either an input time value or the current day
LIB\$FORMAT_DATE_TIME	Formats a date and/or time for output
LIB\$FREE_DATE_TIME_CONTEXT	Frees the date/time context
LIB\$GET_DATE_FORMAT	Returns the user's specified date/time input format
LIB\$GET_MAXIMUM_DATE_LENGTH	Returns the maximum possible length of an output date/time string
LIB\$GET_USERS_LANGUAGE	Returns the user's selected language

# Date/Time Manipulation

**Table 3–1 (Cont.) Date/Time Formatting Routines**

<b>Routine</b>	<b>Function</b>
LIB\$INIT_DATE_TIME_CONTEXT	Initializes the date/time context with a user-specified format
LIB\$MULT_DELTA_TIME	Multiplies a delta time value by an integer scalar value
LIB\$MULTF_DELTA_TIME	Multiplies a delta time value by an F-floating point scalar value
LIB\$SUB_TIMES	Subtracts two quadword times

---

## 3.1 Date/Time Utility Routines

The LIB\$ facility provides date/time utility routines for languages that do not have built-in time and date functions. These routines return information about the current date and time or a date/time specified by the user. The date/time utility routines are as follows:

- Using a string descriptor, LIB\$DATE\_TIME returns the VMS system date and time in the semantics of a string that you provide.
- LIB\$DAY returns the number of days since the system zero date of November 17, 1858. This routine takes one required argument and two optional arguments:
  - The address of a longword to contain the number of days since the system zero date (required).
  - A quadword passed by reference containing a time in system time format to be used instead of the current system time (optional).
  - A longword integer to contain the number of 10-millisecond units since midnight (optional).
- LIB\$DAY\_OF\_WEEK returns the numeric day of the week for an input time value. If the input time value is zero, the current day of the week is returned. The days are numbered 1 through 7: Monday is Day 1, and Sunday is Day 7.

The Run-Time Library also provides a routine, LIB\$SYS\_ASCTIM, that provides a simplified interface between higher-level languages and the \$ASCTIM system service.

---

## 3.2 Date/Time Manipulation Routines

The LIB\$ facility provides several date/time manipulation routines. These routines let you convert an internal time (VMS system time) to an external time, such as “four days,” and vice versa. They also enable you to add, subtract, and multiply dates and times.

The LIB\$ date/time manipulation routines are as follows:

- LIB\$ADD\_TIMES adds two quadword times.

# Date/Time Manipulation

## 3.2 Date/Time Manipulation Routines

- LIB\$CVT\_FROM\_INTERNAL\_TIME converts a VMS standard internal binary time value to an external integer value. The value is converted according to a selected unit of time operation.
- LIB\$CVTF\_FROM\_INTERNAL\_TIME converts a VMS standard internal binary time to an external F-floating point value. The time is converted according to a selected unit of time operation.
- LIB\$CVT\_TO\_INTERNAL\_TIME converts an external integer time value to a VMS standard internal binary time value. The value is converted according to a selected unit of time operation.
- LIB\$CVTF\_TO\_INTERNAL\_TIME converts an external F-floating point time value to an internal binary time value.
- LIB\$CVT\_VECTIM converts a seven-word array (as returned by the \$NUMTIM system service) to a standard VMS format internal time.
- LIB\$MULT\_DELTA\_TIME multiplies a delta time value by an integer scalar value.
- LIB\$MULTF\_DELTA\_TIME multiplies a delta time value by an F-floating point scalar value.
- LIB\$SUB\_TIMES subtracts two quadword times.

---

## 3.3 Date/Time Formatting Routines

The date/time formatting routines allow the user or application programmer to specify input and output formats other than the standard VMS format for dates and times. These include international formats with appropriate language spellings for days and months.

If the desired language is English (the default language) and the desired format is the standard VMS format, then no initialization of logical names is required in order to use the date/time input and output routines. However, if the desired language and format are not the defaults, the system manager (or any user having CMEXEC, SYSNAM and SYSPRV privileges) must initialize the required logicals.

---

### 3.3.1 Date/Time Logical Initialization

**Note:** The initialization steps outlined in this section must be completed before any of the date/time input and output routines can be used with languages and formats other than the defaults.

As an alternative to the standard VMS format, the command procedure SYS\$MANAGER:LIB\$DT\_STARTUP.COM defines several output formats for dates and times. This command procedure must be executed by the system manager prior to the use of any of the Run-Time Library date/time routines for input or output formats other than the default. Ideally, this command procedure should be executed from SYSTARTUP.COM.

In addition to defining the date/time formats, the LIB\$DT\_STARTUP.COM command procedure also defines spellings for date and time elements in languages other than English. If different language spellings are required, the system manager must define the logical name SYS\$LANGUAGES before

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

invoking LIB\$DT\_STARTUP.COM. The translation of SYS\$LANGUAGES is then used to select which languages are defined.

The available languages and their logical names are as follows:

Language	Logical Name
Austrian	AUSTRIAN
Danish	DANISH
Dutch	DUTCH
Finnish	FINNISH
French	FRENCH
French Canadian	CANADIAN
German	GERMAN
Hebrew	HEBREW
Italian	ITALIAN
Norwegian	NORWEGIAN
Portuguese	PORTUGUESE
Spanish	SPANISH
Swedish	SWEDISH
Swiss French	SWISS_FRENCH
Swiss German	SWISS_GERMAN

For example, if the system manager wants the spellings for French, German, and Italian languages to be defined, he or she must define SYS\$LANGUAGES as shown, prior to invoking LIB\$DT\_STARTUP.COM.

```
$ DEFINE SYS$LANGUAGES FRENCH, GERMAN, ITALIAN
```

If the user requires an additional language, for example FINNISH, then the system manager must add FINNISH to the definition of SYS\$LANGUAGES and reexecute the command procedure.

### 3.3.2 Selecting a Format

There are two methods by which date/time input and output formats can be selected:

- The language and format are determined at run time through the translation of the logical names SYS\$LANGUAGE, LIB\$DT\_FORMAT, and LIB\$DT\_INPUT\_FORMAT.
- The language and format are programmable at compile time through the use of the LIB\$INIT\_DATE\_TIME\_CONTEXT routine.

In general, if an application accepts text from a user or formats text for presentation to a user, the logical name method of specifying language and format should be used. In this method, the user assigns equivalence names to the logical names SYS\$LANGUAGE, LIB\$DT\_FORMAT, and LIB\$DT\_INPUT\_FORMAT, thereby selecting the language and input or output format of the date and time at run time.

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

If an application reads text from internal storage or formats text for internal storage or transmission, the language and format should be specified at compile time. If this is the case, the routine `LIB$INIT_DATE_TIME_CONTEXT` is used to specify the language and format of choice.

---

### 3.3.2.1 Run-Time Format Mnemonics

The format mnemonics listed below are used to define both input and output formats at run time. When defining a format, each mnemonic must be preceded by an exclamation mark (!). This exclamation mark signifies that the string represents a format mnemonic; it is not interpreted as part of the format string itself.

#### Date

DO	Day, Zero-filled
DD	Day, No Fill
DB	Day, Blank-filled
WU	Weekday, Uppercase
WAU	Weekday, Abbreviated, Uppercase
WC	Weekday, Capitalized
WAC	Weekday, Abbreviated, Capitalized
WL	Weekday, Lowercase
WAL	Weekday, Abbreviated, Lowercase
MAU	Month, Alphabetic, Uppercase
MAAU	Month, Alphabetic, Abbreviated, Uppercase
MAC	Month, Alphabetic, Capitalized
MAAC	Month, Alphabetic, Abbreviated, Capitalized
MAL	Month, Alphabetic, Lowercase
MAAL	Month, Alphabetic, Abbreviated, Lowercase
MNO	Month, Numeric, Zero-filled
MNM	Month, Numeric, No Fill
MNB	Month, Numeric, Blank-filled
Y4	Year, 4 Digits
Y3	Year, 3 Digits
Y2	Year, 2 Digits
Y1	Year, 1 Digit

#### Time

H04	Hours, Zero-filled, 24-Hour Clock
HH4	Hours, No Fill, 24-Hour Clock
HB4	Hours, Blank-filled, 24-Hour Clock
H02	Hours, Zero-filled, 12-Hour Clock
HH2	Hours, No Fill, 12-Hour Clock
HB2	Hours, Blank-filled, 12-Hour Clock
MO	Minutes, Zero-filled

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

MM	Minutes, No Fill
MB	Minutes, Blank-filled
S0	Seconds, Zero-filled
SS	Seconds, No Fill
SB	Seconds, Blank-filled
C7	Fractional Seconds, 7 Digits
C6	Fractional Seconds, 6 Digits
C5	Fractional Seconds, 5 Digits
C4	Fractional Seconds, 4 Digits
C3	Fractional Seconds, 3 Digits
C2	Fractional Seconds, 2 Digits
C1	Fractional Seconds, 1 Digit
MIU	Meridian Indicator, Uppercase
MIC	Meridian Indicator, Capitalized (mixed case)
MIL	Meridian Indicator, Lowercase

---

### 3.3.2.2 Specifying Formats at Run Time

If an application accepts text from a user or formats text for presentation to a user, the logical name method of specifying language and format should be used. In this method, the user assigns equivalence names to the logical names SYS\$LANGUAGE, LIB\$DT\_FORMAT, and LIB\$DT\_INPUT\_FORMAT, thereby selecting the language and format of the date and time at run time. LIB\$DT\_INPUT\_FORMAT must be defined using the mnemonics listed in Section 3.3.2.1. The possible choices for SYS\$LANGUAGE and LIB\$DT\_FORMAT are defined in the SYS\$MANAGER:LIB\$DT\_STARTUP.COM command procedure that is executed by the system manager prior to using these routines.

The following actions occur when any translation of a logical name fails:

- If the translation of SYS\$LANGUAGE or any logical name relating to text fails, then English is used and a status of LIB\$\_ENGLUSED is returned.
- If the translation of LIB\$DT\_FORMAT, LIB\$DT\_INPUT\_FORMAT, or any logical name relating to format fails, the VMS standard (\$ASCTIM) representation of the date and time is used, that is, dd-mmm-yyyy hh:mm:ss.cc, and a status of LIB\$\_DEFFORUSE is returned.

Since English is the default language and must therefore always be available, English spellings are not taken from logical name translations, but rather are looked up in an internal table.

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

### 3.3.2.2.1 Specifying Input Formats at Run Time

Using the logical name LIB\$DT\_INPUT\_FORMAT, the user can define his or her own input format at run time using the mnemonics listed in Section 3.3.2.1. Once an input format is defined, any dates or times that are input to the application are parsed against this format. For example:

```
$ DEFINE LIB$DT_INPUT_FORMAT -  
_ $ "!MAU !DD, !Y4 !H02:!M0:!S0:!C2 !MIU"
```

A valid input date string would be as follows:

```
JUNE 15, 1988 08:45:06:50 PM
```

If the user has selected a language other than English, then the translation of SYS\$LANGUAGE is used by the parser to recognize alphabetic months and meridian indicators in the selected language.

#### The Input Format String

The input format string used to define the input date/time format must contain at least the first seven of the following eight fields:

- Month (either alphabetic or numeric)
- Day of the month (numeric)
- Year (from 1 to 4 digits)
- Hour (12- or 24-hour clock)
- Minute of the hour
- Second of the minute
- Fractional seconds
- Meridian indicator (required for 12-hour clock; illegal for 24-hour clock)

If the input format string specifies a 24-hour clock, the string will contain only the first seven fields in the above list. If a 12-hour clock is specified, the eighth field (the meridian indicator) is required.

The format string fields must appear in two groups: one for date and one for time (date and time fields cannot be intermixed within a group). For the input format, alphabetic case distinctions and abbreviation-specific codes have no significance. For example, the following format string specifies that the month name will be uppercase and spelled out in full.

```
!MAU !DD, !Y4 !H02:!M0:!S0:!C2 !MIU
```

If the input string corresponding to this format string contains a month name that is abbreviated and lowercased, the parse of the input string still works correctly. For example:

```
feb 25, 1988 04:39:02:55 am
```

If this input string is entered, the parse still recognizes "feb" as the month name and "am" as the meridian indicator, despite the fact that the format string specified both of these fields as uppercased, and the month name as unabbreviated.



# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

### Punctuation in the Format and Input Strings

One important aspect to consider when formatting date/time input strings is punctuation. The punctuation referred to here is the characters that separate the various date/time fields or the date and time groups. Punctuation in these strings is important because it is used as an outline for the parser, allowing the parser to synchronize the input fields to the format fields.

There are three distinct classes of punctuation.

- None

Although it is common to have no punctuation at the beginning or end of an input format string, you may specify a date/time format that additionally has no punctuation between the fields or groups of the format string. If this is the case, the corresponding input string must not have any punctuation between the respective fields or groups, although whitespace (see the next item in this list) may appear at the beginning or end of the input string.

- Whitespace

Whitespace includes any combination of spaces and/or tabs. In the interpretation of the format string, any whitespace is condensed to a single space. When parsing an input string, whitespace is generally noted as synchronizing punctuation and is skipped; however, whitespace is significant in some situations, such as blank-filled numbers.

- Explicit

Explicit punctuation refers to any string of one or more characters that is used as punctuation and is not solely composed of whitespace. Any whitespace appearing within an explicit punctuation string is interpreted literally; in other words, it is not compressed. In the format string, you can use explicit punctuation to denote a particular format and to guide the parser in parsing the input string. In the input string, you can use explicit punctuation to synchronize the parse of the input string against the format string. The explicit punctuation used should not be a subset of the valid input of any field that it precedes or follows.

Punctuation is especially important in providing guidelines for the parser to properly translate the input date/time string.

### Default Date/Time Fields

Punctuation in a date/time string is also useful for specifying which fields you want to omit in order to accept the default values. That is, you can control the parsing of the input string by supplying punctuation without the appropriate field values. If only the punctuation is supplied and no user-supplied default is specified, the value of the omitted field defaults according to the following rules:

- For the date group, the default is the current date.
- For the time group, the default is 00:00:00.00.

Table 3-2 illustrates some examples of input strings (using punctuation to indicate defaulted fields) and their full translations (assuming a current date of 25-FEB-1988 and using the default input format).

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

**Table 3–2 Input String Punctuation and Defaults**

Input	Full Date/Time Input String
31	31-FEB-1988 00:00:00.00
-MAR	25-MAR-1988 00:00:00.00
-SEPTEMBER	25-SEP-1988 00:00:00.00
-1988	25-FEB-1988 00:00:00.00
23:	25-FEB-1988 23:00:00.00
:45:	25-FEB-1988 00:45:00.00
::23	25-FEB-1988 00:00:23.00
.01	25-FEB-1988 00:00:00.01

### 3.3.2.2.2 Specifying Output Formats at Run Time

If the logical name method is used to specify an output format at run time, the translations of the logical names `SYS$LANGUAGE` and `LIB$DT_FORMAT` specify one or more executive mode logicals which in turn must be translated to determine the actual format string. These additional logicals supply such things as the names of the days of the week and the months in the selected language (as determined by `SYS$LANGUAGE`). All of these logicals are predefined, so that a nonprivileged user can select any one of these languages and formats. In addition, a user can create his or her own languages and formats; however, the `CMEXEC`, `SYSNAM`, and `SYSRV` privileges are required.

To select a particular format for a date or time, or both, you must define the `LIB$DT_FORMAT` logical name using the following logicals:

- `LIB$DATE_FORMAT_nnn`, where `nnn` ranges from 001 to 040
- `LIB$TIME_FORMAT_nnn`, where `nnn` ranges from 001 to 020

The order in which these logical names appear in the definition of `LIB$DT_FORMAT` determines the order in which they are output. A single space is inserted into the output string between the two elements, if the definition specifies that both are output. For example:

```
$ DEFINE LIB$DT_FORMAT LIB$DATE_FORMAT_006, LIB$TIME_FORMAT_012
```

The above definition causes the date to be output in the specified format, followed by a space and the time in the specified format, as shown below.

```
13 JAN 88 9:13 AM
```

Table 3–3 lists all predefined date format logical names, their formats, and examples of the output generated using those formats. (The mnemonics used to specify the formats are listed in Section 3.3.2.1.)

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

**Table 3–3 Predefined Output Date Formats**

Date Format Logical	Format	Example
LIB\$DATE_FORMAT_001	IDB-IMAAU-!Y4	13-JAN-1988
LIB\$DATE_FORMAT_002	IDB !MAU !Y4	13 JANUARY 1988
LIB\$DATE_FORMAT_003	!DB.!MAU !Y4	13.JANUARY 1988
LIB\$DATE_FORMAT_004	IDB.!MAU.!Y4	13.JANUARY.1988
LIB\$DATE_FORMAT_005	IDB !MAU !Y2	13 JANUARY 88
LIB\$DATE_FORMAT_006	IDB !MAAU !Y2	13 JAN 88
LIB\$DATE_FORMAT_007	!DB.!MAAU !Y2	13.JAN 88
LIB\$DATE_FORMAT_008	IDB.!MAAU.!Y2	13.JAN.88
LIB\$DATE_FORMAT_009	IDB !MAAU !Y4	13 JAN 1988
LIB\$DATE_FORMAT_010	!DB.!MAAU !Y4	13.JAN 1988
LIB\$DATE_FORMAT_011	!DB.!MAAU.!Y4	13.JAN.1988
LIB\$DATE_FORMAT_012	!MAU !DD, !Y4	JANUARY 13, 1988
LIB\$DATE_FORMAT_013	!MNO!/!DO!/Y2	01/13/88
LIB\$DATE_FORMAT_014	!MNO-!DO-!Y2	01-13-88
LIB\$DATE_FORMAT_015	!MNO.!DO.!Y2	01.13.88
LIB\$DATE_FORMAT_016	!MNO !DO !Y2	01 13 88
LIB\$DATE_FORMAT_017	!DO!/!MNO!/Y2	13/01/88
LIB\$DATE_FORMAT_018	!DO!/!MNO-!Y2	13/01-88
LIB\$DATE_FORMAT_019	!DO-!MNO-!Y2	13-01-88
LIB\$DATE_FORMAT_020	!DO.!MNO.!Y2	13.01.88
LIB\$DATE_FORMAT_021	!DO !MNO !Y2	13 01 88
LIB\$DATE_FORMAT_022	!Y2!/!MNO!/DO	88/01/13
LIB\$DATE_FORMAT_023	!Y2-!MNO-!DO	88-01-13
LIB\$DATE_FORMAT_024	!Y2.!MNO.!DO	88.01.13
LIB\$DATE_FORMAT_025	!Y2 !MNO !DO	88 01 13
LIB\$DATE_FORMAT_026	!Y2!MNO!DO	880113
LIB\$DATE_FORMAT_027	/!Y2.!MNO.!DO	/88.01.13
LIB\$DATE_FORMAT_028	!MNO!/!DO!/Y4	01/13/1988
LIB\$DATE_FORMAT_029	!MNO-!DO-!Y4	01-13-1988
LIB\$DATE_FORMAT_030	!MNO.!DO.!Y4	01.13.1988
LIB\$DATE_FORMAT_031	!MNO !DO !Y4	01 13 1988
LIB\$DATE_FORMAT_032	!DO!/!MNO!/Y4	13/01/1988
LIB\$DATE_FORMAT_033	!DO-!MNO-!Y4	13-01-1988
LIB\$DATE_FORMAT_034	!DO.!MNO.!Y4	13.01.1988
LIB\$DATE_FORMAT_035	!DO !MNO !Y4	13 01 1988
LIB\$DATE_FORMAT_036	!Y4!/!MNO!/DO	1988/01/13
LIB\$DATE_FORMAT_037	!Y4-!MNO-!DO	1988-01-13
LIB\$DATE_FORMAT_038	!Y4.!MNO.!DO	1988.01.13

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

**Table 3–3 (Cont.) Predefined Output Date Formats**

Date Format Logical	Format	Example
LIB\$DATE_FORMAT_039	!Y4 !MNO !DO	1988 01 13
LIB\$DATE_FORMAT_040	!Y4!MNO!DO	19880113

Table 3–4 lists all predefined time format logical names, their formats, and examples of the output generated using those formats.

**Table 3–4 Predefined Output Time Formats**

Time Format Logical	Format	Example
LIB\$TIME_FORMAT_001	!H04:!MO:ISO!C2	09:13:25.14
LIB\$TIME_FORMAT_002	!H04:!MO:ISO	09:13:25
LIB\$TIME_FORMAT_003	!H04.!MO.ISO	09.13.25
LIB\$TIME_FORMAT_004	!H04 !MO ISO	09 13 25
LIB\$TIME_FORMAT_005	!H04:!MO	09:13
LIB\$TIME_FORMAT_006	!H04.!MO	09.13
LIB\$TIME_FORMAT_007	!H04 !MO	09 13
LIB\$TIME_FORMAT_008	!HH4:!MO	9:13
LIB\$TIME_FORMAT_009	!HH4.!MO	9.13
LIB\$TIME_FORMAT_010	!HH4 !MO	9 13
LIB\$TIME_FORMAT_011	!H02:!MO !MIU	09:13 AM
LIB\$TIME_FORMAT_012	!HH2:!MO !MIU	9:13 AM
LIB\$TIME_FORMAT_013	!H04!MO	0913
LIB\$TIME_FORMAT_014	!H04H!MOm	09H13m
LIB\$TIME_FORMAT_015	kl !H04.!MO	kl 09.13
LIB\$TIME_FORMAT_016	!H04H!MO'	09H13'
LIB\$TIME_FORMAT_017	!H04.!MO h	09.13 h
LIB\$TIME_FORMAT_018	h !H04.!MO	h 09.13
LIB\$TIME_FORMAT_019	!HH4 h !MM	9 h 13
LIB\$TIME_FORMAT_020	!HH4 h !MM min !SS s	9 h 13 min 25 s

### 3.3.2.3 Specifying Formats at Compile Time

If an application reads text from internal storage or formats text for internal storage or transmission, the language and format should be specified at compile time. The routine LIB\$INIT\_DATE\_TIME\_CONTEXT allows the user to specify the language and format at compile time by initializing the context area used by LIB\$FORMAT\_DATE\_TIME for output or LIB\$CONVERT\_DATE\_STRING for input with specific strings, instead of through logical name translations. Note that when an application initializes the context area using LIB\$INIT\_DATE\_TIME\_CONTEXT, it expects all required context information to be provided in this way. In other words, it is not expected that some items are preinitialized and other items are gathered through logical name translation.

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

Only one context component can be initialized per call to LIB\$INIT\_DATE\_TIME\_CONTEXT. The available components and their number of elements are listed below. (\_ABB indicates abbreviated versions of the month and weekday names.)

LIB\$_MONTH_NAME	12
LIB\$_MONTH_NAME_ABB	12
LIB\$_FORMAT_MNEMONICS	9
LIB\$_WEEKDAY_NAME	7
LIB\$_WEEKDAY_NAME_ABB	7
LIB\$_RELATIVE_DAY_NAME	3
LIB\$_MERIDIAN_INDICATOR	2
LIB\$_OUTPUT_FORMAT	2
LIB\$_INPUT_FORMAT	1

To specify the actual values for these elements, you must use an initialization string of the following format:

```
"[delim][string-1][delim][string-2][delim]...[delim][string-n][delim]"
```

In this format, **[delim]** is a delimiting character that is not in any of the strings, and **[string-n]** is the spelling of the nth instance of the component.

For example, a string passed to this routine to specify the English spellings of the abbreviated month names might be as follows:

```
"|JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEPT|OCT|NOV|DEC|"
```

The string must contain the exact number of elements for the associated component, otherwise the error LIB\$\_NUMELEMENTS is returned. Note that the string begins and ends with a delimiter. Thus, there is one more delimiter than the number of string elements in the initialization string.

### 3.3.2.3.1 Specifying Input Format Mnemonics at Compile Time

To specify the input format mnemonics at compile time, the user must initialize the component LIB\$\_FORMAT\_MNEMONICS with the appropriate values. Table 3-5 lists the nine fields that must be initialized, in the appropriate order, along with their default (English) values.

**Table 3-5 Legible Format Mnemonics**

Order	Format Field	Legible Mnemonic (Defaults)
1	Year	YYYY
2	Numeric month	MM
3	Numeric day	DD
4	Hours (12- or 24-hour)	HH
5	Minutes	MM
6	Seconds	SS
7	Fractional seconds	CC
8	Meridian indicator	AM/PM
9	Alphabetic month	MONTH

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

For example, the following would be a valid definition of the component LIB\$K\_FORMAT\_MNEMONICS, using English as the natural language:

```
|YYYY|MM|DD|HH|MM|SS|CC|AM/PM|MONTH|
```

If the user were entering the same string using Austrian as the natural language, the definition of the component LIB\$K\_FORMAT\_MNEMONICS would be as follows:

```
|JJJJ|MM|TT|SS|MM|SS|HH| |MONAT|
```

### 3.3.2.3.2 Specifying Output Formats at Compile Time

To specify an output format at compile time, the user must preinitialize the component LIB\$K\_OUTPUT\_FORMAT. Two elements are associated with this output format string. One describes the date format fields, the other the time format fields. The order in which they appear in the string determines the order in which they are output. A single space is inserted into the output stream between the two elements, if the call to LIB\$FORMAT\_DATE\_TIME specifies that both be output. For example:

```
"!|DB-!MAAU-!Y4!|H04:!M0:!S0.!C2!"
```

(These mnemonics are discussed in Section 3.3.2.1.) This format string represents the format used by the \$ASCTIM system service for outputting times. Note that the middle delimiter is replaced by a space in the resultant output.

```
13-JAN-1988 14:54:09:24
```

### 3.3.3 The LIB\$CONVERT\_DATE\_STRING Routine

LIB\$CONVERT\_DATE\_STRING converts an absolute date/time string into a VMS internal format date-time quadword. You can optionally specify which fields of the input string can be defaulted (using the **input-flags** argument), and what the default values should be (using the **defaults** argument). By default, the time fields may be defaulted but the date fields may not. Table 3-2 illustrates some examples of these default values.

The optional **defaulted-fields** argument to LIB\$CONVERT\_DATE\_STRING can be used to determine which input fields were defaulted. That is, the **defaulted-fields** argument is a bit mask in which each set bit indicates that the corresponding field was defaulted in the input date/time string.

If you want to use LIB\$CONVERT\_DATE\_STRING to return the current time as well as the current date, you can call the \$NUMTIM system service and pass the **timbuf** argument, which contains the current date and time, to LIB\$CONVERT\_DATE\_STRING as the **defaults** argument. This tells the LIB\$CONVERT\_DATE\_STRING routine to take the default values for the date and time fields from the 7-word array returned by \$NUMTIM.

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

### 3.3.4 The LIB\$GET\_DATE\_FORMAT Routine

The LIB\$GET\_DATE\_FORMAT routine enables you to retrieve information about the currently selected input format. The string returned by LIB\$GET\_DATE\_FORMAT parallels the currently defined input format string, consisting of the format punctuation (with most whitespace compressed) and "legible" mnemonics representing the various format fields.

Based on the currently defined input date/time format, LIB\$GET\_DATE\_FORMAT returns a string, comprised of the mnemonics that represent the current format. These mnemonics are listed in Table 3-5 in Section 3.3.2.3. The following table illustrates some examples of input format strings and their resultant mnemonic strings (using English as the default language).

Format String	LIB\$GET_DATE_FORMAT Value
IMAU !DD, !Y4 !H04:!M0:!S0:!C2	MONTH DD, YYYY4 HH:MM:SS:CC2
IMNO-!D0-!Y2 !H04:!M0:!S0.!C2	MM-DD-YYYY2 HH:MM:SS.CC2
IMNO-!D0-!Y2 !H02:!M0:!S0.!C2 !MIU	MM/DD/YYYY2 HH:MM:SS.CC2 AM/PM

### 3.3.5 User-Defined Output Formats

In addition to the 40 date output formats and 20 time output formats provided, users can define their own date or time, or both, output formats using the logical names LIB\$DATE\_FORMAT\_nnn and LIB\$TIME\_FORMAT\_nnn, where nnn ranges from 501 to 999. (That is, values of nnn from 001 to 500 are reserved for use by DIGITAL.) The mnemonics used to define output formats are listed in Section 3.3.2.1.

User-defined output formats must be defined as executive mode logicals, and they must be defined in the table LNM\$DT\_FORMAT\_TABLE. These formats are normally defined from the command procedure SYSTARTUP.COM. The following example illustrates the steps required of the system manager to create a particular output format using French as the language:

```
$ DEFINE/EXEC/TABLE=LNM$DT_FORMAT_TABLE LIB$DATE_FORMAT_501 -
_$ "!WL, le !DD !MAL !Y4"
$ DEFINE/EXEC/TABLE=LNM$DT_FORMAT_TABLE LIB$TIME_FORMAT_501 -
_$ "!H04 heures et !M0 minutes"
```

After the system manager defines the desired formats, the user can access them by using the following commands:

```
$ DEFINE SYS$LANGUAGE FRENCH
$ DEFINE LIB$DT_FORMAT LIB$DATE_FORMAT_501, LIB$TIME_FORMAT_501
```

After completing these steps, a program outputting the date and time provides the following results:

```
mardi, le 20 janvier 1988 13 heures et 50 minutes
```

In addition to creating their own date and time formats, users can also define their own language tables (provided they have SYSNAM, SYSPRV, and CMEXEC privileges). To create a language table, a user must define all the logical names required.

# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

For purposes of illustration, the following example defines a portion of the Dutch language table. This table is included in its entirety in the set of predefined languages provided with the international date/time formatting routines.

```
$ CREATE/NAME/PARENT=LNMSYSTEM_DIRECTORY/EXEC/PROT=(S:RWED,G:R,W:R) -
_$ LNM$LANGUAGE_DUTCH
$ DEFINE/EXEC/TABLE=LNMS$LANGUAGE_DUTCH LIB$WEEKDAYS_L -
_$ "maandag", "dinsdag", "woensdag", "donderdag", "vrijdag", -
_$ "zaterdag", "zondag"
$ DEFINE/EXEC/TABLE=LNMS$LANGUAGE_DUTCH LIB$WEEKDAY_ABBREVIATIONS_L -
_$ "maa", "din", "woe", "don", "vri", "zat", "zon"
$ DEFINE/EXEC/TABLE=LNMS$LANGUAGE_DUTCH LIB$MONTHS_L "januari", -
_$ "februari", "maart", "april", "mei", "juni", "juli", "augustus", -
_$ "september", "oktober", "november", "december"
$ DEFINE/EXEC/TABLE=LNMS$LANGUAGE_DUTCH LIB$MONTH_ABBREVIATIONS_L -
_$ "jan", "feb", "mrt", "apr", "mei", "jun", "jul", "aug", "sep", -
_$ "okt", "nov", "dec"
$ DEFINE/EXEC/TABLE=LNMS$LANGUAGE_DUTCH LIB$RELATIVE_DAYS_L -
_$ "gisteren", "vandaag", "morgen"
```

All logicals that are used to build a language are as follows:

### **LIB\$WEEKDAYS\_[U|L|C]**

These logicals supply the names of the weekdays, spelled out in full (either uppercase, lowercase, or mixed case). Weekdays must be defined in order, starting with Monday.

### **LIB\$WEEKDAY\_ABBREVIATIONS\_[U|L|C]**

These logicals supply the abbreviated names of the weekdays (either uppercase, lowercase, or mixed case). Weekday abbreviations must be defined in order, starting with Monday.

### **LIB\$MONTHS\_[U|L|C]**

These logicals supply the names of the months, spelled out in full (either uppercase, lowercase, or mixed case). Months must be defined in order, starting with January.

### **LIB\$MONTH\_ABBREVIATIONS\_[U|L|C]**

These logicals supply the abbreviated names of the months (either uppercase, lowercase, or mixed case). Month abbreviations must be defined in order, starting with January.

### **LIB\$MI\_[U|L|C]**

These logicals supply the spellings for the meridian indicators (either uppercase, lowercase, or mixed case). Meridian indicators must be defined in order; the first indicator represents the hours 0:00:0.0 to 11:59:59.99, and the second indicator represents the hours 12:00:00.00 to 23:59:59.99.

### **LIB\$RELATIVE\_DAYS\_[U|L|C]**

These logicals supply the spellings for the relative days (either uppercase, lowercase, or mixed case). Relative days must be defined in order: yesterday, today, and tomorrow, respectively.

### **LIB\$FORMAT\_MNEMONICS**

This logical supplies the abbreviations for the appropriate format mnemonics. That is, the information supplied in this logical is used to specify a desired input format in the user-defined language. The format mnemonics, along



# Date/Time Manipulation

## 3.3 Date/Time Formatting Routines

with their English values, are listed in the order in which they must be defined.

- 1 Year (YYYY)
- 2 Numeric month (MM)
- 3 Day of the month (DD)
- 4 Hour of the day (HH)
- 5 Minutes of the hour (MM)
- 6 Seconds of the minute (SS)
- 7 Parts of the second (CC)
- 8 Meridian indicator (AM/PM)
- 9 Alphabetic month (MONTH)

The English definition of LIB\$FORMAT\_MNEMONIC is therefore as follows:

```
$ DEFINE/EXEC/TABLE=LNM$LANGUAGE_ENGLISH LIB$FORMAT_MNEMONICS -  
_$ "YYYY", "MM", "DD", "HH", "MM", "SS", "CC", "AM/PM ", "MONTH"
```

# 4

## Condition Handling Routines

This chapter describes the VAX Condition Handling Facility. It is divided into four subsections:

Section 4.1 gives background information on the VAX Condition Handling Facility. It discusses exception conditions, the condition value, and signaling.

Section 4.2 shows you how to use the VAX Condition Handling Facility to write and set up your own condition handlers, initiate the signaling mechanism, and signal application-specific messages.

Section 4.3 shows how to use Run-Time Library condition handling routines. It also shows how to use the Run-Time Library routines that enable and disable the signaling of certain hardware exceptions.

Section 4.4 explains how Run-Time Library routines handle exceptions.

Table 4-1 is a list of Run-Time Library condition handling and signaling routines.

**Table 4-1 Condition Handling and Signaling Routines**

<b>Routine</b>	<b>Function</b>
LIB\$ESTABLISH	Establishes a condition handler
LIB\$REVERT	Deletes a condition handler
LIB\$DEC_OVER	Enables or disables signaling of decimal overflow
LIB\$FLT_UNDER	Enables or disables signaling of floating-point underflow
LIB\$INT_OVER	Enables or disables signaling of integer overflow
LIB\$SIGNAL	Signals an exception condition
LIB\$STOP	Stops execution by using signaling
LIB\$DECODE_FAULT	Analyzes the instruction context for fault
LIB\$FIXUP_FLT	Changes floating-point reserved operand to a specified value
LIB\$MATCH_COND	Matches condition value
LIB\$SIG_TO_STOP	Converts a signaled condition to a condition that cannot be continued
LIB\$SIG_TO_RET	Converts any signal to return status
LIB\$SIM_TRAP	Simulates a floating-point trap

### 4.1

## An Overview of the VAX Condition Handling Facility

VMS provides a set of signaling and condition handling routines and related system services to handle exception conditions. This set of services is called the VAX Condition Handling Facility. The VAX Condition Handling Facility

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

is a part of the common run-time environment of VMS, which includes Run-Time Library routines and other components of VMS.

The VAX Condition Handling Facility provides a single, unified method to enable condition handlers, signal conditions, print error messages, change the error behavior from the system default, and enable or disable detection of certain hardware errors. The RTL and all layered products of VMS use the VAX Condition Handling Facility for condition handling.

See the *Introduction to VMS System Routines* for the functional specification of the VAX Condition Handling Facility.

The following terminology is important in the understanding of the VAX Condition Handling Facility.

### Condition Handling Terminology

#### Exception

An event detected by the hardware or software that changes the normal flow of instruction execution. An exception is a synchronous event caused by the execution of an instruction. When an exception occurs, the processor transfers control by forcing a change in the flow of control from that explicitly indicated in the currently executing process.

Some exceptions are relevant primarily to the current process and normally invoke software in the context of the current process. An integer overflow exception detected by the hardware is an example of an event that is reported to the process. Other exceptions, such as page faults, are handled by the operating system and are transparent to the user.

An exception may also be signaled by a routine (software signaling) by calling the RTL routines LIB\$SIGNAL or LIB\$STOP.

#### Condition

An informational error state which exists when an exception occurs. The term condition is preferred since the term exception implies an error. The term exception condition is used interchangeably with the term condition.

#### Condition Handling

When a condition is detected during the execution of a routine, a signal can be raised by the routine. (See "Signal a condition" under the list of functions below.) The routine is then permitted to respond to the condition. The routine's response is called "handling the condition."

The condition handlers are themselves routines; they have their own call frame. Since they are routines, condition handlers can have condition handlers of their own. This allows condition handlers to field exceptions that might occur in themselves in a modular fashion. A routine may enable a condition handler by placing the address of the condition handler in the first longword of its stack frame.

Parallel mechanisms exist for uniform dispatching of hardware and software exception conditions. Exceptions that are detected and signaled by hardware transfer control to an exception service routine in the executive. Software detected exception conditions are generated by calling the Run-Time Library routines LIB\$SIGNAL or LIB\$STOP. Hardware and software detected exceptions eventually execute the same exception dispatching code. Therefore, a condition handler may handle an exception condition generated by hardware or by software identically.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

The VAX Condition Handling Facility and the related Run-Time Library routines and System Services perform the following functions:

- Establish and call condition handler routines.

You can associate a condition handler for the currently executing routine by specifying an address pointing to the handler, either in the routine's stack frame or in one of the exception vectors. Then, when the routine signals an exception, the VAX Condition Handling Facility calls the condition handler associated with the routine. See Section 4.1.3 for more information about exception vectors. See the *VAX Architecture Reference Manual* for a description of the stack frame and exception vectors. See Figure 4-2 for a sample stack scan for a condition handler.

Most high-level languages provide condition handling statements. BASIC's ON ERROR GOTO and PL/I's ON statements may be used to define condition handlers. If the language does not provide its own condition handling, the RTL routine LIB\$ESTABLISH may be used to enable condition handling.

- Remove an established condition handler routine.

Using LIB\$REVERT, you can remove a condition handler from a routine's stack frame by setting the frame's handler address to zero. If your high-level language provides condition handling statements, you should use them rather than LIB\$REVERT.

- Enable or disable the detection of arithmetic hardware exceptions.

Using Run-Time Library routines, you can enable or disable the signaling of floating-point underflow, integer overflow, and decimal overflow, which are detected by the VAX hardware.

- Signal a condition.

When the hardware detects an exception, such as an integer overflow, a signal is raised at that instruction. A routine may also raise a signal by calling LIB\$SIGNAL or LIB\$STOP. Signals raised by LIB\$SIGNAL allow the condition handler to either terminate or resume the normal flow of the routine. Signals raised by LIB\$STOP require termination of the operation raising the condition. The condition handler will not be allowed to continue from the point of call to LIB\$STOP.

- Display an informational message.

The system establishes default condition handlers before it calls the main program. Because these default condition handlers provide access to the system's standard error messages, the standard method for displaying a message is by signaling if the condition has a severity of informational, warning, or error. See Section 4.1.2 for the definition of the severity field of a condition vector. The system default condition handlers resume execution of the instruction after displaying the messages associated with the signal. If the condition value indicates a severe condition, then the image is exited after the display of the message.

- Display a stack traceback on errors.

The default operations of the LINK and RUN commands provide a system-supplied handler (the traceback handler) to print a symbolic stack traceback. The traceback shows the state of the routine stack at the point where the condition occurred. The traceback information is displayed along with the messages associated with the condition signaled.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

- Compile customer-defined messages.

The VMS Message Utility allows you to define your own exception conditions and the associated messages. Message source files contain the condition values and their associated messages. See Section 4.2.4 for a complete description of how to define your own messages.

- Unwind the stack.

A condition handler can cause a signal to be dismissed and the stack to be unwound to the establisher or caller of the establisher of the condition handler when it returns control to the VAX Condition Handling Facility. During the unwinding operation, the VAX Condition Handling Facility scans the stack. If a condition handler is associated with a frame, the system calls that handler before removing the frame. Calling the condition handlers during the unwind allows a routine to perform cleanup operations specific to a particular application, such as recovering from noncontinuable errors or deallocating resources that were allocated by the routine (such as virtual memory, event flags, and so forth). See Section 4.2.2.3 for a description of the \$UNWIND system service.

- Log error messages to a file.

The Put Message system service (\$PUTMSG) permits any user-written handler to include a message in a listing file. Such message logging can be separate from the default messages the user receives. See Section 4.1.5 for a detailed description of the \$PUTMSG system service.

---

### 4.1.1 Exception Conditions

An exception condition is an event, detected either by hardware or software, that changes the normal flow of instruction execution. Some examples of exception conditions are as follows:

- Arithmetic exception condition in a user-written program detected and signaled by hardware (for example, floating-point overflow)
- Error in a user argument to a Run-Time Library routine detected by software and signaled by calling LIB\$STOP (for example, a negative square root)
- Error in Run-Time Library language-support routine, such as an I/O error or an error in a data type conversion
- RMS success condition, record already locked
- RMS success condition, created file superseded an existing version

There are two standard methods for a DIGITAL- or user-written routine to indicate that an exception condition has occurred:

- 1 Return a completion code to the calling program using the function value mechanism

Most general-purpose Run-Time Library routines indicate exception conditions by returning a condition value in R0. The calling program then tests bit 0 of R0 for success or failure. This method allows better programming structure, because the flow of control can be changed explicitly after the return from each call.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

### 2 Signal the exception condition

A condition may be signaled by calling the RTL routines LIB\$SIGNAL or LIB\$STOP. Any condition handlers that may have been enabled are then called by the VAX Condition Handling Facility. See Section 4.1.4.2 for actions that a condition handler may take. See also Figure 4-2 for the order in which condition handlers are invoked by the VAX Condition Handling Facility.

Exception conditions raised by hardware or software are signaled to the routine identically.

For more details, see Sections 4.1.3 and 4.2.3.

### 4.1.2 The Condition Value

Each exception condition has associated with it a unique 32-bit condition value that identifies the exception condition. Each condition value has a unique system-wide symbol and an associated message. The condition value is used in both methods of indicating exception conditions, returning a status, and signaling.

A condition value includes the following fields:

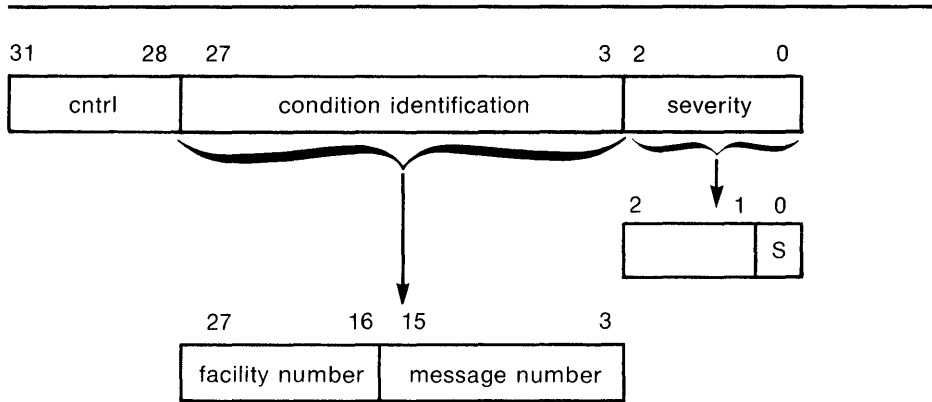
Field	Bits	Meaning
FAC_NO	27 through 16	Indicates the system facility in which the condition occurred. (See Table 2-1 for a list of library facilities and the numbers associated with them.)
MSG_NO	15 through 3	Indicates the particular condition that occurred.
SEVERITY	2 through 0	Indicates whether the condition is a success (bit 0 = 1) or a failure (bit 0 = 0) as well as the severity of the error, if any.

Figure 4-1 shows the format of the condition value.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

Figure 4-1 Format of the Condition Value



ZK-1795-84

### Condition Value Fields

#### severity

The severity code. Bit 0 is set for success (logical true) and clear for failure (logical false); bits 1 and 2 distinguish degrees of success or failure. Bits 0 through 2 represent an unsigned integer, which is interpreted as follows:

Name	Code	Meaning
STS\$_WARNING	0	Warning
STS\$_SUCCESS	1	Success
STS\$_ERROR	2	Error
STS\$_INFO	3	Information
STS\$_SEVERE	4	Severe error
	5,6,7	Reserved to DIGITAL

#### condition identification

Identifies the condition uniquely on a systemwide basis.

#### message number

A status identification, that is, a description of the hardware exception condition that occurred or a software-defined value. Message numbers with bit 15 set are specific to a single facility. Message numbers with bit 15 clear are systemwide status codes.

#### facility number

Identifies the software component generating the condition value. Bit 27 is set for customer facilities and clear for DIGITAL facilities.

#### cntrl

Four control bits. Bit 28 inhibits the message associated with the condition value from being printed by the \$EXIT system service. After using the \$PUTMSG system service to display an error message, the system default handler sets this bit. It is also set in the condition value returned by a routine

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

as a function value, if the routine has also signaled the condition, so that the condition has been either printed or suppressed. Bits 29 through 31 must be zero; they are reserved for DIGITAL.

When a software component completes execution, it returns a condition value in this format. When a severity code of WARNING, ERROR, or SEVERE has been generated, the status code returned describes the nature of the problem. Your program can test this value to change the flow of control or to generate a message. Your program can also generate condition values to be examined by other routines and by the command language interpreter. Condition values defined by customers must set bits 27 and 15 so that these values will not conflict with values defined by DIGITAL.

---

### 4.1.3 Signaling

Signaling can be initiated when hardware or software detects an exception condition. In either case, the exception condition is said to be signaled by the routine in which it occurred. If hardware detects the error, it passes control to a condition dispatcher. If software detects the error, it calls one of the Run-Time Library signal-generating routines: LIB\$SIGNAL or LIB\$STOP. The RTL signal-generating routines pass control to the same condition dispatcher. When LIB\$STOP is called, the severity code is forced to SEVERE, and control cannot return to the routine that signaled the condition. See Section 4.2.2.1 for a description of how a signal may be dismissed and how normal execution from the point of the exception condition may be continued.

When a routine signals, it passes to the VAX Condition Handling Facility the condition value associated with the exception condition, as well as optional arguments that can be passed to a condition handler. The VAX Condition Handling Facility uses these arguments to build two data structures on the stack: the signal argument vector and the mechanism argument vector. These two vectors become the arguments that the VAX Condition Handling Facility passes to condition handlers.

- The signal argument vector contains the information describing the nature of the exception condition.
- The mechanism argument vector describes the state of the process at the time the exception condition occurred.

These argument vectors are described in detail in Sections 4.1.3.1 and 4.1.3.2.

After the signal and mechanism argument vectors are set up, the VAX Condition Handling Facility searches for enabled condition handlers. A condition handler is a separate routine that has been associated with a routine in order to take a specific action when an exception condition occurs. The VAX Condition Handling Facility searches for condition handlers to handle the exception condition, beginning with the primary exception vector of the access mode in which the exception condition occurred. If this vector contains the address of a handler, it is called. If the address is zero, or if the handler resignals, then the VAX Condition Handling Facility repeats the process with the secondary exception vector. Enabling vectored handlers is discussed in detail in the *Introduction to VMS System Routines*. Because the exception vectors are allocated in static storage, they are not generally used by modular routines.



# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

If neither the primary nor secondary vectored handlers handle the exception condition by continuing program execution, then the VAX Condition Handling Facility looks for stack frame condition handlers. It looks for the address of a condition handler in the first longword of the routine stack frame where the exception condition occurred. At this point, several actions are possible, depending on the contents of the first longword:

- If the address is zero, then this routine has not set up a condition handler. In this case, the VAX Condition Handling Facility continues the stack scan by moving to the previous stack frame (that is, the stack frame of the calling routine).
- If the address is not zero, then a condition handler is present. The VAX Condition Handling Facility then calls this handler, which may resignal, continue, or unwind (see Section 4.1.4.2).

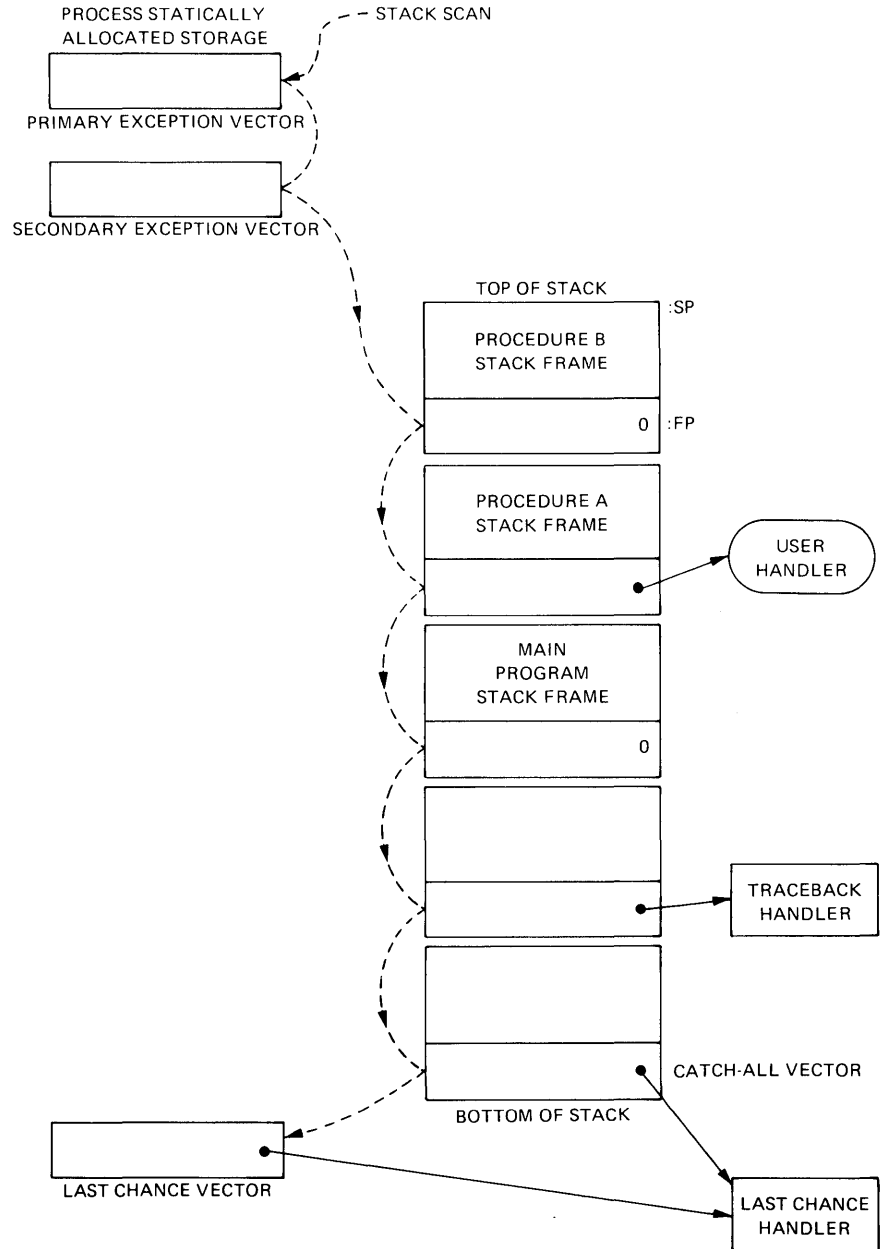
The VAX Condition Handling Facility searches for and calls condition handlers from each frame on the stack until the frame pointer is zero (indicating the end of the call sequence). In that case, the VAX Condition Handling Facility calls the vectored catch-all handler, which displays an error message and causes the program to exit. Note that normally the frame containing the stack catch-all handler is the end of the calling sequence or the bottom of the stack. Section 4.1.4 explains the possible actions of default and user condition handlers in more detail.

Figure 4-2 illustrates a stack scan for condition handlers in which the main program calls routine A, which then calls routine B. A stack scan is initiated when a hardware exception condition occurs or a call is made to LIB\$SIGNAL or LIB\$STOP.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

Figure 4-2 Sample Stack Scan for Condition Handlers



ZK-1935-84

### 4.1.3.1 Signal Argument Vector

The signal argument vector contains information describing the nature of the hardware or software condition. Figure 4-3 illustrates the open-ended structure of the signal argument vector, which can be from 3 to 257 longwords in length.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

Figure 4-3 Format of the Signal Argument Vector

	MACRO and BLISS	High-Level Languages
n = additional longwords	CHF\$ <u>L</u> _SIG_ARGS	SIGARGS(1)
Condition value	CHF\$ <u>L</u> _SIG_NAME	SIGARGS(2)
Optional additional arguments making up one or more message sequences		
PC		SIGARGS(n)
PSL		SIGARGS(n + 1)

ZK-1963-84

### Fields of the Signal Argument Vector

#### SIGARGS(1)

An unsigned integer (n) designating the number of longwords that follow in the vector, including PC and PSL. For example, the first entry of a 4-longword vector would contain a 3.

#### SIGARGS(2)

A condition value indicating the condition being signaled. Handlers should always check to see if the condition is the one that they expect by examining the STS\$V\_COND\_ID field of the condition value (bits 27:3). Bits 2:0 are the severity field. Bits 31:28 are control bits; they may have been changed by an intervening handler and so should not be included in the comparison. You can use the Run-Time Library routine LIB\$MATCH\_COND to match the correct fields. If the condition is not expected, the handler should resignal by returning FALSE (bit 0 = 0).

#### SIGARGS(3 to n-1)

Optional arguments that provide additional information about the condition. These arguments consist of one or more message sequences. The format of a message sequence is described in Section 4.1.5.

#### SIGARGS(n)

The Program Counter (PC) of the next instruction to be executed if any handler (including the system-supplied handlers) returns with the status SS\$\_CONTINUE. For hardware faults, the PC is that of the instruction that caused the fault. For hardware traps, the PC is that of the instruction following the one that caused the trap. For conditions signaled by calling LIB\$SIGNAL or LIB\$STOP, the PC is the location following the CALLS or CALLG instruction. See the *VAX Architecture Reference Manual* for a detailed description of faults and traps.

#### SIGARGS(n+1)

The Processor Status Longword (PSL) of the program at the time that the condition was signaled. See the *VAX Architecture Reference Manual* for more about the Processor Status Longword.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

**Note:** LIB\$SIGNAL and LIB\$STOP copy the variable-length argument list passed by the caller. Then, before calling a condition handler, they append the PC and PSL entries to the end of the list.

The formats for some conditions signaled by the operating system and the Run-Time Library are shown in Figures 4-4 and 4-5.

**Figure 4-4 Signal Argument Vector for the Reserved Operand Error Conditions**

---

3	Additional longwords
SS\$_ROPRAND	Condition value
PC	PC of instruction causing fault
PSL	

ZK-1964-84

---

**Figure 4-5 Signal Argument Vector for RTL Mathematics Routine Errors**

---

5	Additional longwords
MTH\$_abcmnoxyz	Math condition value
1	Number of FAO args
Caller's PC	PC following JSB or CALL
PC	PC following call to LIB\$SIGNAL
PSL	

ZK-1965-84

---

The caller's PC is the PC following the calling program's JSB or CALL to the mathematics routine that detected the error. The PC is that following the call to LIB\$SIGNAL.

---

### 4.1.3.2 Mechanism Argument Vector

The mechanism argument vector contains all of the information describing the state of the process at the time of the hardware or software signaled condition. Figure 4-6 illustrates a mechanism argument vector, which is a 5-longword vector.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

Figure 4–6 Format of a Mechanism Argument Vector

	MACRO and BLISS	High-Level Languages
4 = additional longwords	CHF\$L_MCH_ARGS	MCHARGS(1)
Frame	CHF\$L_MCH_FRAME	MCHARGS(2)
Depth	CHF\$L_MCH_DEPTH	MCHARGS(3)
Saved R0	CHF\$L_MCH_SAVR0	MCHARGS(4)
Saved R1	CHF\$L_MCH_SAVR1	MCHARGS(5)

ZK-1966-84

### Fields of the Mechanism Argument Vector

#### MCHARGS(1)

An unsigned integer indicating the number of longwords that follow in the vector. Currently, this number is always four.

#### MCHARGS(2)

The address of the stack frame of the routine that established the handler being called. This address can be used as a base from which to reference the local stack-allocated storage of the establisher, as long as the restrictions on the handler's use of storage are observed (see Section 4.2.2).

#### MCHARGS(3)

The stack depth, which is the number of stack frames between the establisher of the condition handler and the frame in which the condition was signaled. To ensure that calls to LIB\$SIGNAL and LIB\$STOP appear as similar as possible to hardware exception conditions, the call to LIB\$SIGNAL or LIB\$STOP is not included in the depth.

If the routine that contained the hardware exception condition or called LIB\$SIGNAL or LIB\$STOP also handled the exception condition, then the depth is zero; if the exception condition occurred in a called routine and its caller handled the exception condition, then the depth is 1; and so on. If a system service signals an exception condition, a handler established by the immediate caller is entered with a depth of 1.

The depth is -2 for a handler established using the primary exception vector, -1 for the secondary vector, and -3 for the last-chance vector.

#### MCHARGS(4) and MCHARGS(5)

Copies of the contents of registers R0 and R1 at the time of the exception condition or the call to LIB\$SIGNAL or LIB\$STOP. When execution continues or a stack unwind occurs, these values are restored to R0 and R1. Thus a handler can modify these values to change the function value returned to a caller.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

---

### 4.1.4 Condition Handlers

When a routine is activated, the first longword in its stack frame is set to zero. This longword is reserved to contain an address pointing to another routine called the condition handler. If an exception condition is signaled during the execution of the routine, the VAX Condition Handling Facility uses the address in the first longword of the frame to call the associated condition handler. The arguments passed to the condition handling routine are the signal and mechanism argument vectors described in Sections 4.1.3.1 and 4.1.3.2.

There are various types of condition handlers that can be called for a given routine:

- User-supplied condition handlers

You can write your own condition handler and set up its address in the stack frame of your routine using the Run-Time Library routine LIB\$ESTABLISH or the mechanism supplied by your language.

- Language-supplied condition handlers

Many high-level languages provide a means for setting up handlers that are global to a single routine. For example, BASIC's ON ERROR GOTO and PL/I's ON statement provide language-specific condition handlers. If your language provides a condition handling mechanism, you should always use it. If you also try to establish a condition handler using LIB\$ESTABLISH, the two methods of handling exception conditions conflict, and the results are unpredictable.

- System default condition handlers

VMS provides a set of default condition handlers. These take over if there are no other condition handler addresses on the stack, or if all the previous condition handlers have passed on (resignaled) the indication of the exception condition.

**Note:** Do not use LIB\$ESTABLISH to set up a condition handler in a language that defines its own condition handling, such as BASIC, COBOL, and PL/I.

---

#### 4.1.4.1 Default Condition Handlers

VMS establishes the following default condition handlers each time a new image is started. The default handlers are shown in the order they are encountered when VMS processes a signal. These three handlers are the only handlers that output error messages.

- Traceback handler

The traceback handler is established on the stack after the catch-all handler. This enables the traceback handler to get control first. This handler performs three functions in the order shown:

- 1 Outputs an error message using the Put Message (\$PUTMSG) system service. \$PUTMSG formats the message using the Formatted ASCII Output (\$FAO) system service and sends the message to the devices SYS\$ERROR and SYS\$OUTPUT (if it differs from SYS\$ERROR).
- 2 Issues a symbolic traceback, which shows the state of the routine stack at the time of the exception condition.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

- 3 Decides whether to continue executing the image or to force an exit based on the severity field of the condition value.

Severity	Error Type	Action
1	SUCCESS	Continue
3	INFO	Continue
0	WARNING	Continue
2	ERROR	Continue
4	SEVERE	Exit

You can eliminate the traceback handler at link time by using the /NOTRACEBACK qualifier in the link command.

- Catch-all handler

VMS establishes the catch-all handler in the first stack frame and thus calls it last. This handler performs the same functions as the traceback handler, except for the stack traceback. That is, it issues an error message and decides whether to continue execution. It is called only if you link with the /NOTRACEBACK qualifier.

- Last-chance handler

VMS establishes the last-chance handler with a system exception vector. In most cases, this vector contains the address of the catch-all handler, so that these two handlers are actually the same. The last-chance handler is called only if the stack is invalid or all the handlers on the stack have resigned. If the debugger is present, the debugger's own last-chance handler will replace the system last-chance handler.

### 4.1.4.2 Possible Condition Handler Actions

When a condition handler returns control to the VAX Condition Handling Facility, the VAX Condition Handling Facility takes one of the following three actions, depending on the value returned by the condition handler.

- Resignal the condition ( $R0 = SS\_RESIGNAL$  or  $R0 <0> = 0$ ).

When a condition handler resignals, it passes control back to the VAX Condition Handling Facility with a status code of  $SS\_RESIGNAL$ . This indicates that the handler was unable to deal with the particular exception condition, and it is passing the indication on to other handlers. A handler can alter the severity of the signal before resignaling.

Section 4.2.2.2 contains more information about resignaling.

- Continue from the point of the signal ( $R0 = SS\_CONTINUE$  or  $R0 <0> = 1$ ).

When a condition handler returns the status  $SS\_CONTINUE$ , the VAX Condition Handling Facility returns control to the routine that signaled the exception condition, beginning at the instruction that initiated signaling (in the case of faults) or at the instruction following the one that initiated signaling (in the case of traps). A condition handler cannot continue if the exception condition was signaled by calling  $LIB\$STOP$ .

Section 4.2.2.1 contains more information about continuing.

- Unwind the call stack and dismiss the signal.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

When a condition handler has already called \$UNWIND, any return status from the condition handler is ignored by the VAX Condition Handling Facility. The VAX Condition Handling Facility now unwinds the stack.

Unwinding the routine call stack removes call frames, starting with the frame in which the condition occurred, and returns control to an earlier routine in the calling sequence. You can unwind the stack whether the condition was detected by hardware or signaled using LIB\$SIGNAL or LIB\$STOP. Unwinding is the only way to continue execution after a call to LIB\$STOP. A condition handler unwinds by calling the Unwind (\$UNWIND) system service.

Section 4.2.2.3 shows how to write a condition handler that unwinds the call stack.

### 4.1.4.3 Interaction Between Default and User-Supplied Handlers

Several results are possible after a routine signals, depending on several factors, such as the severity of the error, the method of generating the signal, and the action of the condition handlers you have defined and the default handlers. Given the severity of the condition and the method of signaling, Table 4-2 lists all combinations of interaction between user condition handlers and default condition handlers.

**Table 4-2 Interaction Between Handlers and Default Handlers**

Severity of condition	User handler specifies CONTINUE	User handler specifies UNWIND	Default handler gets control	No handler found (bad stack)
<b>Exception condition is signaled by a call to LIB\$SIGNAL or detected by hardware</b>				
WARNING, INFO, or ERROR	RETURN	UNWIND	Issue condition message RETURN	Call last-chance handler EXIT
SEVERE	RETURN	UNWIND	Issue condition message EXIT	Call last-chance handler EXIT
<b>Exception condition is signaled by a call to LIB\$STOP</b>				
LIB\$STOP forces severity to SEVERE	Message: "Attempt to continue from stop" EXIT	UNWIND	Issue condition message EXIT	Call last-chance handler EXIT

ZK-4257-85



# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

### 4.1.5 Displaying Messages

The standard format for a VMS message is as follows:

`%FACILITY-L-IDENT, message-text`

<b>FACILITY</b>	Abbreviated name of the software component that issued the message.
<b>L</b>	Indicator showing the severity level of the exception condition that caused the message.
<b>IDENT</b>	Symbol of up to nine characters representing the message.
<code>message-text</code>	Brief definition of the cause of the message.

The message can also include up to 255 formatted-ASCII-output (FAO) arguments. These arguments can be used to display variable information about the condition that caused the message. In the following examples, the file specification is an FAO argument:

`%TYPE-W-OPENIN, error opening _DBO:[FOSTER]AUTHOR.DAT; as input`

Signaling provides a consistent and unified method for displaying messages. This section describes how the VAX Condition Handling Facility translates the original signal into a human-readable message.

When any software detects an exception condition, it signals the exception condition to the user by calling `LIB$SIGNAL` or `LIB$STOP`. The signaling routine passes a signal argument list to these Run-Time Library routines. This signal argument list is made up of the condition value and a set of optional arguments that provide information to condition handlers. Signaling is used to signal exception conditions generated by DIGITAL-supplied software.

You can use the signaling mechanism to signal messages specific to your application. Further, you can chain your own message onto a system message. For more information, see Section 4.2.4.

`LIB$SIGNAL` and `LIB$STOP` copy the signal argument list and use it to create the signal argument vector. The signal argument vector serves as part of the input to the user-established handlers and the system default handlers.

If all intervening handlers have resigned, the system default handlers take control. The system-supplied default handlers are the only handlers that should actually issue messages, whether the exception conditions are signaled by DIGITAL-supplied software or your own programs. That is, a routine should signal exception conditions, rather than issuing its own messages. In this way, other applications can call the routine and override its signal in order to change the messages. Further, this technique makes the choice of formatting details and wording centralized and consistent.

The system default handlers pass the signal argument vector to the Put Message (`$PUTMSG`) system service. `$PUTMSG` formats and displays the information in the signal argument vector.

`$PUTMSG` performs the following steps:

- 1 Interprets the signal argument vector as a series of one or more message sequences. Each message sequence starts with a 32-bit, systemwide condition value that identifies a message in the system message file.

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

- 2 Obtains the text of the message using the Get Message (\$GETMSG) system service. The message text definition is actually a SYS\$FAO control string. It may contain embedded FAO (formatted ASCII output) directives. These directives determine how the FAO arguments in the signal argument vector are formatted. (See the \$FAO system service in the *VMS System Services Reference Manual*.)
- 3 Calls \$FAO to format the message, substituting the values from the signal argument list.
- 4 Issues the message on device SYS\$OUTPUT. If SYS\$ERROR is different from SYS\$OUTPUT, and the severity field in the condition value is not SUCCESS, \$PUTMSG also issues the message on device SYS\$ERROR.

The *VMS System Services Reference Manual* describes \$PUTMSG in detail.

Each message sequence in the signal argument list produces one line of output. Figure 4-7 illustrates the three possible message sequence formats.

**Figure 4-7 Formats of Message Sequences**

---

### No FAO (Formatted ASCII Output) arguments

Condition value
-----------------

Note that a condition value of zero results in no message.

### Variable number of FAO arguments

Condition value
FAO_count
FAO arg 1
FAO arg 2
.
.
.
FAO arg n

Condition value

Number of FAO arguments

### VAX-11 RMS error with STV (Status Value)

VAX-11 RMS Condition Value (STS)
Associated status value (STV)

Condition value

One FAO argument or  
SS\$... condition value

ZK-1967-84

# Condition Handling Routines

## 4.1 An Overview of the VAX Condition Handling Facility

RMS system services return two related completion values, the completion code and the associated status value. The completion code is returned in R0, using the function value mechanism. The same value is also placed in Completion Status Code field of the RMS File Access Block or Record Access Block associated with the file (FAB\$L\_STS or RAB\$L\_STS). The status value is returned in the Status Value field of the same FAB or RAB (FAB\$L\_STV or RAB\$L\_STV). The meaning of this secondary value is based on the corresponding STS (Completion Status Code) value. Its meaning could be any of the following:

- An operating system condition value of the form SS\$\_ . . .
- An RMS value, such as the size of a record which exceeds the buffer
- Zero

Rather than have each calling program determine the meaning of the STV value, \$PUTMSG performs the necessary processing. Therefore, this STV value must always be passed in place of the FAO argument count. In other words, a RMS message sequence always consists of two arguments (passed by immediate value): an STS value and an STV value.

---

### 4.1.6 Multiple Active Signals

A signal is said to be active until the routine that signaled regains control or until the stack is unwound or the image exits. A second signal can occur while a condition handler or a routine it has called is executing. This situation is called multiple active signals. When this situation occurs, the stack scan is not performed in the usual way. Instead, the frames that were searched while processing all of the previous exception conditions are skipped when the current exception condition is processed. This is done in order to avoid recursively reentering a routine which is not reentrant. For example, FORTRAN code is typically not recursively reentrant. If a FORTRAN handler were called while another activation of that handler was still going, the results would be unpredictable.

The modified search routine is best illustrated with an example. Assume the following calling sequence:

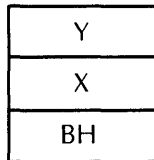
- 1 Routine A calls routine B, which calls routine C.
- 2 Routine C signals an exception condition (signal S), and the handler for routine C (CH) resignals.
- 3 Control passes to BH, the handler for routine B. The call frame for handler BH is located on top of the signal and mechanism arrays for signal S. The saved frame pointer in the call frame for BH points to the frame for routine C.
- 4 BH calls routine X; routine X calls routine Y.
- 5 Routine Y signals a second exception condition (signal T). Figure 4-8 illustrates the stack contents after the second exception condition is signaled.

# Condition Handling Routines

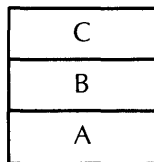
## 4.1 An Overview of the VAX Condition Handling Facility

**Figure 4-8** Stack After Second Exception Condition Is Signaled

< Signal T >



< Signal S >



ZK-1968-84

Normally, the VAX Condition Handling Facility searches all currently active frames for condition handlers, including B and C. If this happens, however, BH is called again. At this point, you skip the condition handlers that have already been called. Thus, the search for condition handlers should proceed in the following order:

YH  
XH  
BHH (the handler for routine B's handler)  
AH

- 6 The search now continues in its usual fashion. The VAX Condition Handling Facility examines the primary and secondary exception vectors, then frames Y, X, and BH. Thus handlers YH, XH, and BHH are called. Assume that these handlers resignal.
- 7 The VAX Condition Handling Facility now skips the frames that have already been searched and resumes the search for condition handlers in routine A's frame. The depths that are passed to handlers as a result of this modified search are 0 for YH, 1 for XH, 2 for BHH, and 3 for AH.

Because of the possibility of multiple active signals, you should be careful if you use an exception vector to establish a condition handler. Vectored handlers are called, not skipped, each time an exception occurs.

# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

---

### 4.2 Using the VAX Condition Handling Facility

This section shows you how to build condition handling into your program. This process involves one or more of the following steps:

- Establishing the handler in the stack frame of your routine
- Writing a condition handling routine or choosing one of the Run-Time Library routines that handles exception conditions
- Including a call to a Run-Time Library signal generating routine
- Using the message utility to define your own exception conditions
- Including a call to the \$PUTMSG system service to modify or log the system error message

---

#### 4.2.1 Establishing a Condition Handler

A condition handler is established when its address has been placed in the first longword of the stack frame of the routine with which it is to be associated. To establish a condition handler, call the Run-Time Library routine LIB\$ESTABLISH, using the name of the handler as an argument. LIB\$ESTABLISH returns as a function value the address of the former handler established for the routine or zero if no handler existed. Some languages provide a mechanism for setting up a condition handler. If so, use the language mechanism rather than LIB\$ESTABLISH.

The new condition handler remains in effect for your routine until you call LIB\$REVERT or control returns to the caller of the caller of LIB\$ESTABLISH. Once this happens, you must call LIB\$ESTABLISH again if the same (or a new) condition handler is to be associated with the caller of LIB\$ESTABLISH.

The Run-Time Library provides several condition handlers and routines that a condition handler can call. These routines take care of several common exception conditions. Section 4.3 describes these routines.

---

#### 4.2.2 Writing a Condition Handler

You can write a condition handler to take action when an exception condition is signaled. When the exception condition occurs, the VAX Condition Handling Facility sets up the signal argument vector and mechanism argument vector and begins the search for a condition handler. Therefore, your condition handling routine must declare variables to contain the two argument vectors. Further, the handler must indicate the action to be taken when it returns to the VAX Condition Handling Facility. The handler uses its function value to do this. Thus, the calling sequence for your condition handler has the following format:

```
handler      signal-args ,mechanism-args
```

##### **signal-args**

The address of a vector of longwords indicating the nature of the condition. See Section 4.1.3.1, Signal Argument Vector, for a detailed description.

# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

### **mechanism-args**

The address of a vector of longwords that indicate the state of the process at the time of the signal. See Section 4.1.3.2, Mechanism Argument Vector, for more details.

### **result**

A condition value. Success (bit 0 = 1) causes execution to continue at the PC and failure (bit 0 = 0) causes the condition to be resigaled. That is, the system resumes the search for other handlers. If the handler calls the Unwind (\$UNWIND) system service, the return value is ignored and the stack is unwound. See Section 4.2.2.3.

Handlers can modify the contents of either the **signal-args** vector or the **mechanism-args** vector.

In order to protect compiler optimization, a condition handler and any routines that it calls can only reference arguments explicitly passed to handlers. They cannot reference COMMON or other external storage, nor can they reference local storage in the routine that established the handler, unless the compiler considers the storage to be volatile. Compilers that do not adhere to this rule must ensure that any variables referenced by the handler are always kept in memory, not in a register.

As mentioned previously, a condition handler can take one of three actions:

- Continue execution
- Resignal the exception condition and resume the stack scanning operation
- Call \$UNWIND to unwind the call stack to an earlier frame

The sections that follow show how to write condition handlers to perform these three operations.

### **4.2.2.1 Continuing Execution**

To continue execution from the instruction following the signal, with no error messages or traceback, the handler returns with the function value SS\$\_CONTINUE (bit 0 = 1). If, however, the condition was signaled with a call to LIB\$STOP, the SS\$\_CONTINUE return status causes an error message (ATTEMPT TO CONTINUE FROM STOP) and the image exits. The only way to continue from a call to LIB\$STOP is for the condition handler to request a stack unwind.

If execution is to continue after a hardware fault (such as a reserved operand fault), the condition handler must correct the cause of the condition before returning the function value SS\$\_CONTINUE or requesting a stack unwind. Otherwise, the instruction that caused the fault will be executed again.

**Note:** On most VAX machines, hardware floating-point traps have been changed to hardware faults. If you still want floating-point exception conditions to be treated as traps, use LIB\$SIM\_TRAP to simulate the action of floating-point traps.

# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

---

### 4.2.2.2 Resignaling

Condition handlers check for specific errors. If the signaled condition is not one of the expected errors, the handler resignals. That is, it returns control to the VAX Condition Handling Facility with the function value `SS$_RESIGNAL` (with bit 0 clear). To alter the severity of the signal, the handler modifies the low three bits of the condition value and resignals.

For an example of resignaling, see Section 4.1.6.

---

### 4.2.2.3 Unwinding the Call Stack

A condition handler can dismiss the signal by calling the system service `$UNWIND`. The stack unwind is initiated when a condition handler that has called `$UNWIND` returns to VAX Condition Handling Facility. Unwinding the routine call stack removes call frames and returns control to the routine specified in the **depth** argument of `$UNWIND`. You can unwind the stack whether the condition was detected by hardware or signaled by `LIB$SIGNAL` or `LIB$STOP`. Unwinding the stack is the only way to continue execution after a call to `LIB$STOP`.

Normally, when a condition handler dismisses the signal, control is returned to the establisher or caller of the establisher of the condition handler and normal execution resumes.

Your condition handler unwinds by calling the Unwind (`$UNWIND`) system service. The *VMS System Services Reference Manual* describes `$UNWIND` in detail. The Program Counter (PC) of the continuation point may also be specified to the `$UNWIND` service.

When a condition handler calls `$UNWIND` and returns to the VAX Condition Handling Facility, the return status is ignored and the following actions occur:

- 1 The VAX Condition Handling Facility scans each call frame in the call stack, beginning with the one in which the condition occurred, to see if a condition handler has been established.
- 2 If a handler has been established, the VAX Condition Handling Facility calls it, placing the value `SS$_UNWIND` in the condition value field of the signal argument vector. This indicates that the stack is being unwound.
- 3 The handler performs the cleanup operations required by the routine that established it. For example, the handler should deallocate any processwide resources that have been allocated. Then, the handler returns control to the VAX Condition Handling Facility.
- 4 After control returns to VAX Condition Handling Facility or if there is no handler established for the routine, the routine's stack frame is removed from the stack.
- 5 When a condition handler is called during the unwinding operation, the condition handler must not generate a new signal. A new signal would result in unpredictable behavior.

# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

The number of frames removed depends on the arguments passed to \$UNWIND.

- If both arguments are omitted, the stack is unwound to the caller of the routine that established the handler.
  - 1 Routine A calls routine B.
  - 2 Routine B establishes handler C.
  - 3 An exception condition occurs during the execution of routine B.
  - 4 The VAX Condition Handling Facility calls handler C.
  - 5 Handler C calls \$UNWIND with no arguments and returns.

A call to \$UNWIND which contains no **depth** argument is a request by the handler to unwind to the caller of its establisher once the handler returns. Therefore, when handler C returns, handler C is called again with an SS\$\_UNWIND condition code. When handler C returns, the call frame for routine B is removed and control is resumed at the point of call to routine B at routine A.
- If the first argument **depth** is specified, this argument specifies the number of frames to be removed.
- If the second argument **new-PC** is specified, it indicates the address of the instruction to which control will return after the unwind.

An unwind to the caller of the establisher of the condition handler causes the frame of the routine that established the handler to be unwound. In this case, the handler will be called twice: once when the VAX Condition Handling Facility calls it to handle the raised condition, and again when the frame in which the handler is established is removed. In the example above, handler C is called twice.

Handlers established by the primary, secondary, or last-chance vectors are not called, since they are not removed during an unwind operation.

While it is unwinding the stack, the VAX Condition Handling Facility ignores any function value returned by a condition handler. For this reason, a handler cannot both resignal and unwind. Thus, the only way for a handler to both issue a message and perform an unwind is to call LIB\$SIGNAL and then call \$UNWIND. If your program calls \$UNWIND before calling LIB\$SIGNAL, the result is unpredictable.

When the VAX Condition Handling Facility calls the condition handler established for each frame during unwind, the call is of the standard form, described in Section 4.2.1. The arguments passed to the condition handler (the signal and mechanism argument vectors) are shown in Figure 4-9.

If the handler is to specify the function value of the last function to be unwound, it should modify the saved copies of R0 and R1 (CHF\$\_MCH\_SAVR0 and CHF\$\_MCH\_SAVR1) in the mechanism argument vector. R0 and R1 are restored from the mechanism argument vector at the end of the unwind.



# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

Figure 4–9 Arguments Passed to Condition Handler During Unwind

---

signal-args	
1	Number of following longwords
SS\$_UNWIND	Condition value

mechanism-args	
4	Number of following longwords
Establisher's FP	Frame Pointer of the frame that established the handler
0	Always zero
R0	R0 that unwind will restore
R1	R1 that unwind will restore

ZK-1969-84

---

### 4.2.3 Generating Signals

When software detects an exception condition, it normally calls one of the Run-Time Library signal-generating routines, LIB\$SIGNAL or LIB\$STOP, to initiate the signaling mechanism. This call indicates to the calling program that the exception condition has occurred. Your program can also call one of these routines explicitly to indicate an exception condition. Furthermore, some languages have built-in methods for signaling errors specific to the language.

When your program wants to issue a message and allow execution to continue after handling the condition, it calls the standard routine, LIB\$SIGNAL. The calling sequence for LIB\$SIGNAL is the following:

```
LIB$SIGNAL condition-value1 [,number1] [,FAO-arg1...FAO-argn1]
           [,condition-value2] [,number2] [,FAO-arg2...FAO-argn2]
```

Only the **condition-value1** argument must be specified; other arguments are optional. The **number1** argument, if specified, contains the number of FAO arguments that will be associated with **condition-value1**. The **condition-value2** argument is optional; it may be specified with or without the **number2** or **FAO-arg2** arguments. The **number2** argument, if specified, contains the number of FAO arguments that will be associated with **condition-value2**. You may specify **condition-value3**, **condition-value4**, **condition-value5**, and so on, along with their corresponding **number** and **FAO** arguments.

For further information about the arguments to LIB\$SIGNAL, refer to the description of LIB\$SIGNAL in the reference section of this manual.

# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

### condition-value

VMS Usage:           cond\_value  
type:                 longword (unsigned)  
access:               read only  
mechanism:            by value

VAX 32-bit condition value. The **condition-value** argument is an unsigned longword that contains this condition value. Section 4.1.2 explains the format of a VAX condition value.

### number

VMS Usage:           longword\_signed  
type:                 longword integer (signed)  
access:               read only  
mechanism:            by value

Number of FAO arguments associated with the condition value. The **number** argument is a signed longword integer that contains this number. If omitted or specified as zero, no FAO arguments follow.

The maximum number of FAO arguments specified must not exceed 253. See Sections 4.1.5 and 4.2.4 for more information about FAO arguments.

### FAO-arg

VMS Usage:           varying\_arg  
type:                 unspecified  
access:               read only  
mechanism:            by value

Additional FAO (formatted ASCII output) arguments that are associated with the specified condition value. The **FAO-arg** argument is the address of a signed longword integer or a character string that contains these additional FAO arguments. Section 4.1.5 explains the message format.

When your program wants to issue a message and stop execution unconditionally, it calls LIB\$STOP. The calling sequence for LIB\$STOP is as follows:

```
LIB$STOP condition-value1 [,number1] [,FAO-arg1...,FAO-argn1]
          [,condition-value2] [,number2] [,FAO-arg2...,FAO-argn2]
```

Only the **condition-value1** argument must be specified; other arguments are optional. The **number1** argument, if specified, contains the number of FAO arguments that will be associated with **condition-value1**. The **condition-value2** argument is optional; it may be specified with or without the **number2** or **FAO-arg2** arguments. The **number2** argument, if specified, contains the number of FAO arguments that will be associated with **condition-value2**. You may specify **condition-value3**, **condition-value4**, **condition-value5**, and so on, along with their corresponding **number** and **FAO** arguments.

For further information about the arguments to LIB\$STOP, refer to the description of LIB\$STOP in the LIB\$ Reference Section of this manual.

# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

In both cases, **condition-value** indicates the condition that is being signaled. However, LIB\$STOP always sets the severity of **condition-value** to SEVERE before proceeding with the stack-scanning operation.

The FAO arguments describes the details of the exception condition. These are the same arguments that are passed to the VAX Condition Handling Facility as part of the signal argument vector. The system default condition handlers pass them to \$PUTMSG, which uses them to issue a system message.

Unlike most routines, LIB\$SIGNAL and LIB\$STOP preserve R0 and R1 as well as the other registers. Therefore, a call to LIB\$SIGNAL allows the debugger to display the entire state of the process at the time of the exception condition. This is useful for debugging checks and gathering statistics.

The behavior of LIB\$SIGNAL is the same as that of the exception dispatcher that performs the stack scan after hardware detects an exception condition. That is, the system scans the stack in the same way and the same arguments are passed to each condition handler. This allows a user to write a single condition handler to detect both hardware and software conditions.

### 4.2.4 Signaling User-Defined Messages

Section 4.1.5 explains how the VAX Condition Handling Facility displays messages. The signal argument list passed to LIB\$SIGNAL or LIB\$STOP can be seen as one or more message sequences. Each message sequence consists of a condition value, an FAO count, which specifies the number of FAO arguments to come, and the FAO arguments themselves. (The FAO count is omitted in the case of SYSTEM and RMS messages.) The message text definition itself is actually a SYS\$FAO control string, which may contain embedded \$FAO directives. The *VMS System Services Reference Manual* describes the Formatted ASCII Output (\$FAO) system service in detail.

The VMS Message Utility is provided for compiling message sequences specific to your application. When you have defined an exception condition and used the VMS Message Utility to associate a message with that exception condition, your program can call LIB\$SIGNAL or LIB\$STOP to signal the exception condition. Then the system default condition handlers will display your error message in the standard VMS format.

To use the Message Utility, follow these steps:

- 1 Create a source file that specifies the information used in messages, message codes, and message symbols.
- 2 Use the MESSAGE command to compile this source file.
- 3 Link the resulting object module, either by itself or with another object module containing a program.
- 4 Run your program so that the messages are accessed, either directly or through the use of pointers.

See the description of the VMS Message Utility in the *VMS Message Utility Manual*.

You signal a message that is defined in a message source file by calling LIB\$SIGNAL or LIB\$STOP, as for any software-detected exception condition.

# Condition Handling Routines

## 4.2 Using the VAX Condition Handling Facility

A signal argument list may contain one or more condition values and FAO arguments. Each condition value and its FAO arguments is “chained” to the next condition value and its FAO arguments. You can use chained messages to provide more specific information about the exception condition being signaled, along with a general message.

The following message source file defines an exception condition PROG\_\_FAIGETMEM:

```
.FACILITY      PROG,1 /PREFIX=PROG__
.SEVERITY      FATAL
.BASE         100
FAIGETMEM     <failed to get !UL bytes of memory>/FAO_COUNT=1
.END
```

This source file sets up the exception message as follows:

- The .FACILITY directive specifies the facility, PROG, and its number, 1. It also adds the /PREFIX qualifier to determine the prefix to be used in the message.
- The .SEVERITY directive specifies that PROG\_\_FAIGETMEM is a fatal exception condition. That is, the SEVERITY field in the condition value for PROG\_\_FAIGETMEM is set to SEVERE (bits 0:3 = 4).
- The BASE directive specifies that the condition identification numbers in the PROG facility will begin with 100.
- FAIGETMEM is the symbol name. This name is combined with the prefix defined in the facility definition to make the message symbol. The message symbol becomes the symbolic name for the condition value.
- The text in angle brackets is the message text. This is actually a SYS\$FAO control string. When \$PUTMSG calls the \$FAO system service to format the message, \$FAO includes the FAO argument from the signal argument vector and formats the argument according to the embedded FAO directive (!UL).
- The .END statement terminates the list of messages for the PROG facility.

### 4.2.5 Logging Error Messages to a File

You can write a condition handler to obtain a copy of a system error message text and write the message into an auxiliary file, such as a listing file. In this way, you can receive identical messages at the terminal (or batch log file) and in the auxiliary file.

To log messages, you must write a condition handler and an action subroutine. Your handler calls the Put Message (\$PUTMSG) system service explicitly. The operation of \$PUTMSG is described in Section 4.1.5. The handler passes to \$PUTMSG the signal argument vector and the address of the action subroutine. \$PUTMSG passes to the action subroutine the address of a string descriptor which contains the length and address of the formatted message. The action subroutine can scan the message, copy it into a log file, or both.

# Condition Handling Routines

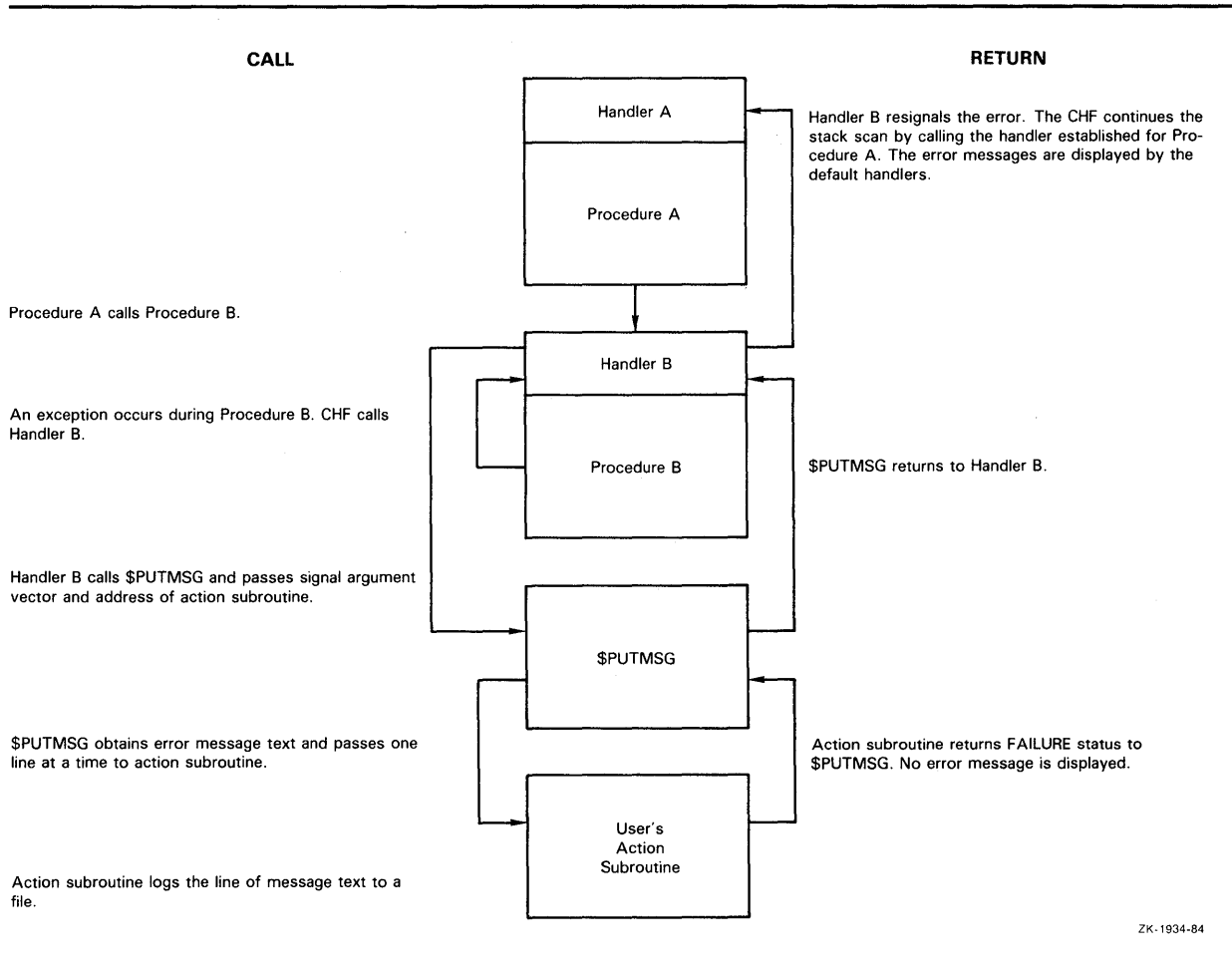
## 4.2 Using the VAX Condition Handling Facility

It is important to keep the display messages centralized and consistent. Thus, you should use only \$PUTMSG to display or log system error messages. Further, because the system default handlers call \$PUTMSG to display error messages, your handlers should avoid displaying the error messages. There are two ways to do this:

- 1 Your handler should not call \$PUTMSG directly to display an error message. Instead, your handler should resignal the error. This allows other calling routines to change or suppress the message, or recover from the error. The system default condition handlers will display the message.
- 2 If the action subroutine that you supply to \$PUTMSG returns a success code, \$PUTMSG displays the error message on SYS\$OUTPUT or SYS\$ERROR, or both. Thus, your action routine should process the message and then return a failure code, so that \$PUTMSG will not display the message at this point.

Figure 4-10 shows the sequence of events involved in calling \$PUTMSG to log an error message to a file.

**Figure 4-10 Using a Condition Handler to Log an Error Message**



ZK-1934-84

# Condition Handling Routines

## 4.3 Run-Time Library Condition Handling Routines

---

### 4.3 Run-Time Library Condition Handling Routines

The Run-Time Library provides several routines that can be established as condition handlers or called from a condition handler to handle signaled exception conditions. This section shows how to use these routines.

---

#### 4.3.1 Convert a Floating-Point Fault to a Floating-Point Trap

A trap is an exception condition that is signaled after the instruction that caused it has finished executing. A fault is an exception condition that is signaled during the execution of the instruction. When a trap is signaled, the PC (Program Counter) in the signal argument vector points to the next instruction after the one that caused the exception condition. When a fault is signaled, the PC in the signal argument vector points to the instruction that caused the exception condition. See the *VAX Architecture Reference Manual* for more information about faults and traps.

LIB\$SIM\_TRAP can be established as a condition handler, or called from a condition handler, to convert a floating-point fault to a floating-point trap. After LIB\$SIM\_TRAP is called, the PC points to the instruction after the one that caused the exception condition. Thus your program can continue execution without fixing up the original condition. LIB\$SIM\_TRAP intercepts only floating overflow, underflow, and divide-by-zero faults.

---

#### 4.3.2 Change a Signal to a Return Status

When it is preferable to detect errors by signaling, but the calling routine expects a returned status, LIB\$SIG\_TO\_RET may be used by the routine that signals. LIB\$SIG\_TO\_RET is a condition handler that converts any signaled condition to a return status. The status is returned to the caller of the routine that established LIB\$SIG\_TO\_RET. You may establish LIB\$SIG\_TO\_RET as a condition handler by specifying it in a call to LIB\$ESTABLISH.

LIB\$SIG\_TO\_RET may also be called from another condition handler. LIB\$SIG\_TO\_RET is called from a condition handler, the signaled condition is returned as a function value to the caller of the establisher of that handler when the handler returns to the VAX Condition Handling Facility. When a signaled exception condition occurs, LIB\$SIG\_TO\_RET routine does the following:

- Places the signaled condition value in the image of R0 that is saved as part of the mechanism argument vector.
- Calls the Unwind (\$UNWIND) system service with the default arguments. After returning from LIB\$SIG\_TO\_RET (when it is established as a condition handler) or after returning from the condition handler that called LIB\$SIG\_TO\_RET (when LIB\$SIG\_TO\_RET is called from a condition handler), the stack unwinds to the caller of the routine that established the handler.

Your calling routine can now test R0, as if the called routine had returned a status, and specify an error recovery action.

# Condition Handling Routines

## 4.3 Run-Time Library Condition Handling Routines

---

### 4.3.3 Change a Signal to a Stop

LIB\$SIG\_TO\_STOP causes a signal to appear as though it had been signaled by a call to LIB\$STOP.

LIB\$SIG\_TO\_STOP may be enabled as a condition handler for a routine or it may be called from a condition handler. When a signal is generated by LIB\$STOP, the severity code is forced to SEVERE and control cannot return to the routine that signaled the condition. See Section 4.2.2.1 for a description of continuing normal execution after a signal.

---

### 4.3.4 Match Condition Values

LIB\$MATCH\_COND checks for a match between two condition values to allow a program to branch according to the condition found. If no match is found, the routine returns zero. The routine only matches the condition identification field (STS\$V\_COND\_ID) of the condition value; it ignores the control bits and the severity field. If the facility-specific bit (STS\$V\_FAC\_SP = bit 15) is clear in cond-val (meaning that the condition value is systemwide), LIB\$MATCH\_COND ignores the facility code field (STS\$V\_FAC\_NO = bits 27:17) and compares only the STS\$V\_MSG\_ID fields (bits 15:3).

---

### 4.3.5 Correct a Reserved Operand Condition

After a signal of SS\$\_ROPRAND during a floating-point instruction, LIB\$FIXUP\_FLT finds the operand and changes it from -0.0 to a new value or to +0.0.

---

### 4.3.6 Decode the Instruction That Generated a Fault

LIB\$DECODE\_FAULT locates the operands for an instruction that caused a fault and passes the information to a user action routine. When called from a condition handler, LIB\$DECODE\_FAULT locates all the operands and calls an action routine that you supply. Your action routine performs the steps necessary to handle the exception condition and returns control to LIB\$DECODE\_FAULT. LIB\$DECODE\_FAULT then restores the operands and the environment, as modified by the action routine, and continues execution of the instruction.

---

## 4.4 How Run-Time Library Routines Handle Exceptions

Most general-purpose Run-Time Library routines handle errors by returning a status in R0. In some cases, however, exceptions that occur during the execution of a Run-Time Library routine are signaled. This section tells how Run-Time Library routines signal exception conditions.

Some calls to the Run-Time Library do not or cannot specify an action to be taken. In this case, the Run-Time Library will signal the proper exception condition using the VAX signaling mechanism.

# Condition Handling Routines

## 4.4 How Run-Time Library Routines Handle Exceptions

In order to maintain modularity, the Run-Time Library does not use exception vectors, which are processwide data locations. Thus the Run-Time Library itself does not establish handlers using the primary, secondary, or last-chance exception vectors.

### 4.4.1 Exception Conditions Signaled from Mathematics Routines

Mathematics routines return function values in register R0 or registers R0/R1, unless the return values are larger than 64 bits. For this reason, mathematics routines cannot use R0 to return a completion status and must signal all errors. In addition, all mathematics routines signal an error specific to the MTH\$ facility, rather than a general hardware error.

#### 4.4.1.1 Integer Overflow and Floating-Point Overflow

Although the hardware normally detects integer overflow and floating-point overflow errors, Run-Time Library mathematics routines are programmed with a software check to trap these conditions before the hardware signaling process can occur. This means that they call LIB\$SIGNAL, instead of allowing the hardware to initiate signaling.

The software check is needed because JSB routines cannot set up condition handlers. The check permits the JSB mathematics routines to add an extra stack frame so that the error message and stack traceback will appear as if a CALL instruction had been performed. Because of the software check, JSB routines will not cause a hardware exception condition even when the calling program has enabled the detection of integer overflow. On the other hand, floating-point overflow detection is always enabled and cannot be disabled.

If an integer or floating-point overflow occurs during a CALL or a JSB routine, the routine signals a mathematics-specific error such as MTH\$\_FLOOVEMAT (Floating Overflow in Math Library) by calling LIB\$SIGNAL explicitly.

#### 4.4.1.2 Floating-Point Underflow

All mathematics routines are programmed to avoid floating-point underflow conditions. Software checks are made to determine if a floating-point underflow condition would occur. If so, the software makes an additional check:

- If the immediate calling program (CALL or JSB) has enabled floating-point underflow traps, a mathematics-specific error condition is signaled.
- Otherwise, the result is corrected to zero and execution continues with no error condition.

The user can enable or disable floating-point underflow detection at run time by calling the routine LIB\$FLT\_UNDER.



# Condition Handling Routines

## 4.4 How Run-Time Library Routines Handle Exceptions

### 4.4.2 Overflow/Underflow Detection Enabling Routines

You can use the following Run-Time Library routines to enable or disable the signaling of decimal overflow, floating-point underflow, and integer overflow:

- LIB\$DEC\_OVER enables or disables the reporting of decimal overflow.
- LIB\$FLT\_UNDER enables or disables the reporting of floating-point underflow.
- LIB\$INT\_OVER enables or disables the reporting of integer overflow.

You cannot disable the signaling of integer divide-by-zero, floating-point overflow, and floating-point or decimal divide-by-zero.

When the signaling of a hardware condition is enabled, the occurrence of the exception condition causes VMS to signal the condition as a severe error. When the signaling of a hardware condition is disabled, the occurrence of the condition is ignored and the processor executes the next instruction in the sequence.

The signaling of overflow and underflow detection is enabled independently for each routine activation, since the call instruction saves the state of the calling program's hardware enable operations in the stack and then initializes the enable operations for the called routine. A return instruction restores the calling program's enable operations.

These Run-Time Library routines are intended primarily for higher-level languages, since you can achieve the same effect in MACRO with the single Bit Set PSW (BISPSW) or Bit Clear PSW (BICPSW) instructions.

These routines allow you to enable and disable detection of decimal overflow, floating-point underflow, and integer overflow for a portion of your routine's execution. Note that the BASIC and FORTRAN compilers provide a compile-time qualifier that permits you to enable or disable integer overflow for your entire routine.

# 5

---

## Memory Allocation Routines

This chapter discusses the Run-Time Library routines that perform dynamic memory allocation functions.

The topics covered in this chapter include the following:

- Section 5.1 provides an overview of dynamic memory allocation.
- Section 5.2 describes routines for allocating and freeing pages of memory.
- Section 5.3 introduces the concept of zones and describes the routines used to create, delete, find, show, reset, and verify zones.
- Section 5.4 describes the routines used to allocate and free variable-sized blocks of memory.
- Section 5.5 discusses the algorithms that can be used to allocate memory within a zone.
- Section 5.6 introduces user-defined zones that can be used to implement additional allocation algorithms or to monitor the behavior of other zones.
- Section 5.7 describes interactions between the Run-Time Library memory allocation routines and high-level language memory allocation facilities, as well as interactions with other Run-Time Library routines.
- Section 5.8 discusses interactions between the Run-Time Library memory allocation routines and VMS memory management system services.

---

### 5.1 Overview

Sophisticated software systems must often create and manage complex data structures. In these systems, the size and number of elements are not always known in advance. You can tailor the memory allocation for these elements by using *dynamic memory allocation*. By managing the memory allocation, you can avoid allocating fixed tables that may be too large or too small for your program. To help you manage memory, the Run-Time Library and the VMS operating system provide a hierarchy of routines and services for memory management.

---

#### 5.1.1 Virtual Address Space

The virtual address space of an executing program consists of the following three regions:

**1** A process program region

The process program region is also referred to as P0 space. P0 space contains the instructions and data for the current image.

Your program can dynamically allocate storage in the process program region by calling Run-Time Library dynamic memory allocation routines or VMS system services.

# Memory Allocation Routines

## 5.1 Overview

### 2 A process control region

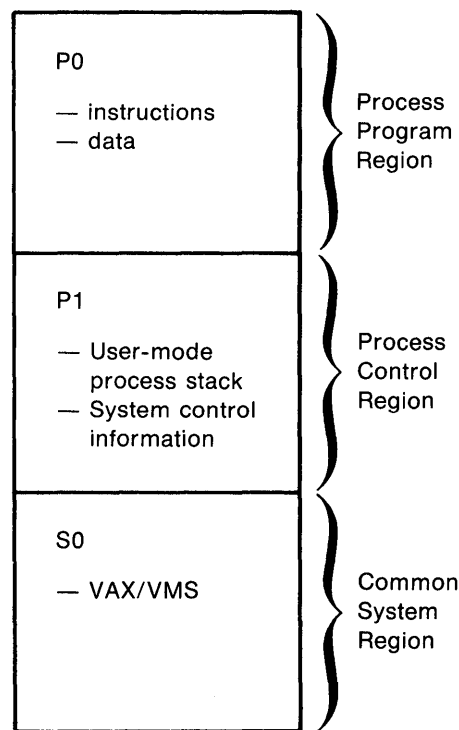
The process control region is also referred to as P1 space. P1 space contains system control information and the user-mode process stack. The user mode stack expands as necessary toward the lower-addressed end of P1 space.

### 3 A common system region

The common system region is also referred to as S0 space. S0 space contains the VMS operating system. Your program cannot allocate or free memory within the common system region from the user access mode.

A summary of these regions appears in Figure 5-1.

**Figure 5-1 Virtual Address Overview**



ZK-4145-85

---

### 5.1.2 Memory Allocation Routines

Memory allocation routines allow you to allocate and free storage within the virtual address space available to your process.

# Memory Allocation Routines

## 5.1 Overview

This section describes three levels of memory allocation routines. These levels are as follows:

### 1 Memory management system services

The memory management system services comprise the lowest level of memory allocation routines. These services include, but are not limited to, the following:

- \$EXPREG (Expand Region)
- \$CRETVA (Create Virtual Address Space)
- \$DELTVA (Delete Virtual Address Space)
- \$CRMPSC (Create and Map Section)
- \$MGBLSC (Map Global Section)
- \$DGBLSC (Delete Global Section)

For most cases in which a system service is used for memory allocation, the Expand Region system service (\$EXPREG) is used to create 512-byte pages of virtual memory.

### 2 Run-Time Library page management routines

The Run-Time Library page management routines LIB\$GET\_VM\_PAGE and LIB\$FREE\_VM\_PAGE provide a convenient mechanism for allocating and freeing pages of memory.

These routines maintain a processwide pool of free pages. If no unallocated pages are available when LIB\$GET\_VM\_PAGE is called, it calls \$EXPREG to create the required pages in the program region (P0 space).

### 3 Run-Time Library heap management routines

The Run-Time Library heap management routines LIB\$GET\_VM and LIB\$FREE\_VM provide a mechanism for allocating and freeing blocks of memory of arbitrary size.

LIB\$CREATE\_VM\_ZONE, LIB\$CREATE\_USER\_VM\_ZONE, LIB\$DELETE\_VM\_ZONE, LIB\$FIND\_VM\_ZONE, LIB\$RESET\_VM\_ZONE, LIB\$SHOW\_VM\_ZONE and LIB\$VERIFY\_VM\_ZONE are heap management routines based on the concept of zones. A zone is a logically independent memory pool or subheap. Refer to Section 5.3 for more information about zones.

If no unallocated block is available to satisfy a call to LIB\$GET\_VM, it calls LIB\$GET\_VM\_PAGE to allocate additional pages.

Modular application programs can call routines in any or all levels of the hierarchy, depending on the kinds of services the application program needs. The basic rule that must be observed when using multiple levels of the hierarchy is as follows:

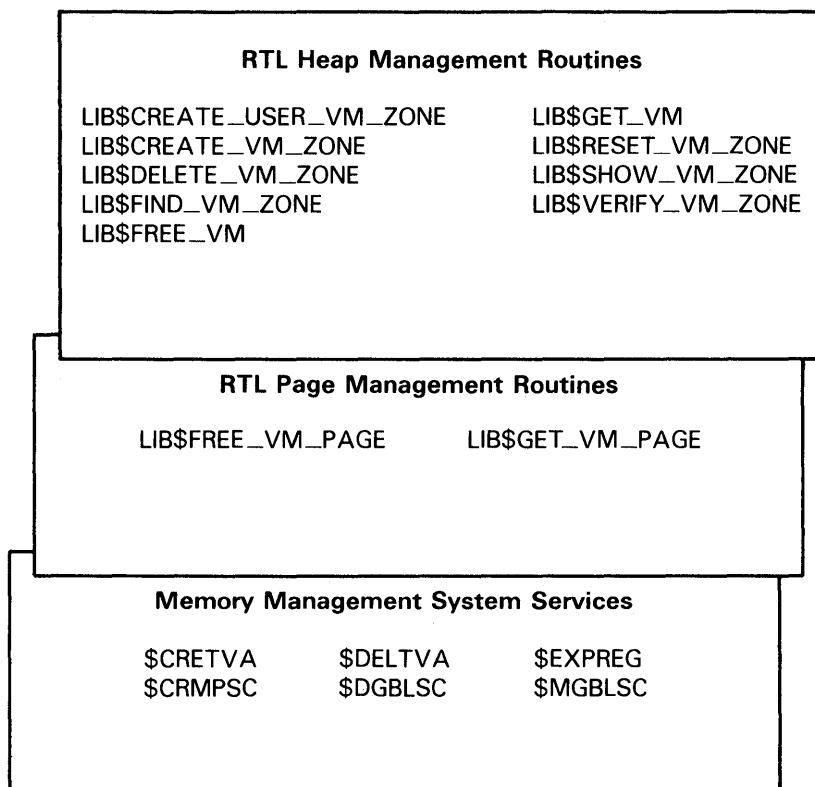
- Memory that is allocated by an allocation routine at one level of the hierarchy must be freed by calling a deallocation routine at the same level of the hierarchy. For example, if you allocated a page of memory by calling LIB\$GET\_VM\_PAGE, you can free it only by calling LIB\$FREE\_VM\_PAGE.

# Memory Allocation Routines

## 5.1 Overview

Figure 5-2 shows the three levels of memory allocation routines.

**Figure 5-2 Hierarchy of Memory Management Routines**



ZK-4146-85

## 5.2 Allocating and Freeing Pages

The Run-Time Library page management routines `LIB$GET_VM_PAGE` and `LIB$FREE_VM_PAGE` provide a flexible mechanism for allocating and freeing pages of memory. In general, modular routines should use these routines rather than direct system service calls to manage pages of memory. The page management routines maintain a processwide pool of free pages and automatically reuse free pages. If your program calls system services directly, it must do the bookkeeping to keep track of free pages.

`LIB$GET_VM_PAGE` and `LIB$FREE_VM_PAGE` are fully reentrant. They can be called by code running at AST level or in an Ada multitasking environment.

All pages allocated by `LIB$GET_VM_PAGE` are created with user-mode read-write access, even if the call to `LIB$GET_VM_PAGE` is made from a more privileged access mode.

# Memory Allocation Routines

## 5.2 Allocating and Freeing Pages

LIB\$GET\_VM\_PAGE and LIB\$FREE\_VM\_PAGE are designed for request sizes ranging from one page to a few hundred pages. If you are using very large request sizes (over 1000 contiguous pages in a single request), the bitmap allocation method that is used may cause fragmentation of your virtual address space. For very large request sizes, you should use direct calls to \$EXPREG.

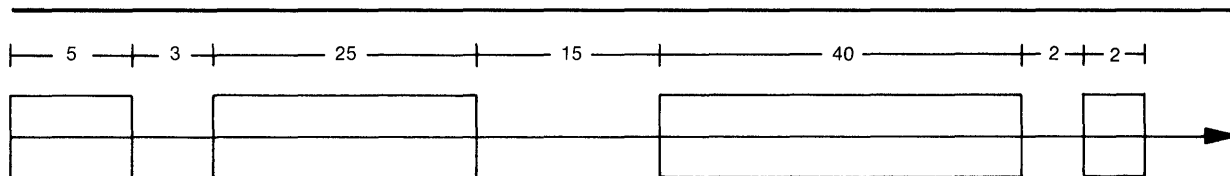
The rules for using LIB\$GET\_VM\_PAGE and LIB\$FREE\_VM\_PAGE are as follows:

- Any memory you free by calling LIB\$FREE\_VM\_PAGE must have been allocated by a previous call to LIB\$GET\_VM\_PAGE. You cannot allocate memory by calling \$EXPREG or \$CRETVA and then free it using LIB\$FREE\_VM\_PAGE.
- All memory allocated by LIB\$GET\_VM\_PAGE is page aligned; that is, the low-order 9 bits of the address are all zero. All memory freed by LIB\$FREE\_VM\_PAGE must also be page aligned; an error status will be returned if you attempt to free a block of memory that is not page aligned.
- You can free a smaller group of pages than you allocated. That is, if you allocated a group of four contiguous pages by a single call to LIB\$GET\_VM\_PAGE, you can free the memory by using several calls to LIB\$FREE\_VM\_PAGE; for example, free one page, two pages, and one page.
- You can combine contiguous groups of pages that were allocated by several calls to LIB\$GET\_VM\_PAGE into one group of pages that are freed by a single call to LIB\$FREE\_VM\_PAGE. Before doing this, however, you must compare the addresses to ensure that the pages you are combining are indeed contiguous. Of course, you must ensure that a routine only frees pages that it has previously allocated and that it still owns.
- Be especially careful that you do not attempt to free a set of pages twice. It is possible that you may free a set of pages in one routine and reallocate those same pages from another routine. If the first routine then deallocates those pages a second time, any information that was stored in them by the second routine is lost. Because the pages are still allocated to your program (even though to a different routine), this type of programming mistake will not generate an error.
- The contents of memory allocated by LIB\$GET\_VM\_PAGE are unpredictable. Your program must assign values to all locations that it uses.
- You should try to minimize the number of request sizes your program uses to avoid fragmentation of the free page pool. This concept is shown in Figure 5-3.

# Memory Allocation Routines

## 5.2 Allocating and Freeing Pages

Figure 5–3 Memory Fragmentation



ZK-4150-85

The straight line running across Figure 5–3 represents the memory allocated to your program. The blocks represent memory that has already been allocated. At this point, if you request 16 pages, memory will have to be allocated at the far right end of the memory line shown in this figure, even though there are 20 free pages before that point. You cannot use 16 of these 20 pages because the 20 free pages are “fragmented” into groups of 15, 3, and 2 pages.

Fragmentation is discussed further in Section 5.3.1.

## 5.3 Zones

The Run-Time Library heap management routines `LIB$GET_VM` and `LIB$FREE_VM` are based on the concept of zones. A zone is a logically independent memory pool or subheap; a program may use several zones to structure its heap memory management.

You create a zone with specified attributes by calling the routine `LIB$CREATE_VM_ZONE`. `LIB$CREATE_VM_ZONE` returns a zone-id value that you can use in subsequent calls to the routines `LIB$GET_VM` and `LIB$FREE_VM`. When you no longer need the zone, you can delete the zone and free all the memory it controls by a single call to `LIB$DELETE_VM_ZONE`.

If you want a program to deal with each VM zone created during the invocation, including those created outside of the program, you can call `LIB$FIND_VM_ZONE`. At each call, `LIB$FIND_VM_ZONE` scans the heap management database and returns the zone identifier of the next valid zone.

`LIB$SHOW_VM_ZONE` returns formatted information about a specified zone, detailing such information as the zone’s name, characteristics, and areas, and then passes the information to the specified or default action routine. `LIB$VERIFY_VM_ZONE` verifies the zone header and scans all of the queues and lists maintained in the zone header.

If you call `LIB$GET_VM` to allocate memory from a zone and the zone has no free memory to satisfy the request, `LIB$GET_VM` calls `LIB$GET_VM_PAGE` to allocate a block of contiguous pages for the zone. Each such block of contiguous pages is called an area. You control the number of pages in an area by specifying the area extension size attribute when you create the zone.

# Memory Allocation Routines

## 5.3 Zones

The systematic use of zones provides the following benefits:

- Structuring heap memory management

Data structures in your program may have different lifetimes or dynamic scopes. Some structures may continue to grow during the entire execution of your program, while others exist for a very short time and are then discarded by the program. You can create a separate zone in which you allocate a particular type of short-lived structure. When the program no longer needs any of those structures, you can delete all of them in a single operation by calling `LIB$DELETE_VM_ZONE`.

- Program locality

Program locality is a characteristic of a program that indicates the distance between the references and virtual memory over a period of time. A program with a high degree of program locality does not refer to many widely scattered virtual addresses in a short period of time. Maintaining a high degree of program locality reduces the number of page faults and improves program performance.

It is important to minimize the number of page faults to obtain best performance in a virtual memory system like VMS. For example, if your program creates and searches a symbol table, you can reduce the number of page faults incurred by the search operation by using as few pages as possible to hold all the symbol table entries. If you allocate symbol table entries and other items unrelated to the symbol table in the same zone, each page of the symbol table will contain both symbol table entries and other items. Because of the extra unrelated entries, the symbol table will take up more pages than it actually needs. A search of the symbol table will then access more pages and performance will be lower as a result. You may be able to reduce the number of page faults by creating a separate symbol table zone, so that pages that contain symbol table entries do not contain any unrelated items.

- Specialized allocation algorithms

No single memory allocation algorithm is ideal for all applications. Section 5.5 describes the Run-Time Library memory allocation algorithms and their performance characteristics so that you can select an appropriate algorithm for each zone that you create.

- Performance tuning

You can specify a number of attributes that affect performance when you create a zone. The allocation algorithm you select can have a significant effect on performance. By specifying the allocation block size, you can improve performance and reduce fragmentation within the zone at the cost of some extra memory. Boundary tags can also be used to improve the speed of `LIB$FREE_VM` at the cost of some extra memory. Boundary tags are further discussed in Section 5.3.1.



# Memory Allocation Routines

## 5.3 Zones

---

### 5.3.1 Zone Attributes

You can specify a number of zone attributes when you call `LIB$CREATE_VM_ZONE` to create the zone. The attributes that you specify are permanent; that is, you cannot change the attribute values. They remain constant until you delete the zone. Each zone that you create can have a different set of attribute values. Thus you can tailor each zone to optimize program locality, execution time, and memory usage.

This section describes each of the zone attributes, suggested values for the attribute, and the effects of the attribute on execution time and memory usage. If you do not specify a complete set of attribute values, `LIB$CREATE_VM_ZONE` will provide defaults for many of the attributes. More detailed information about argument names and the encoding of arguments is given in the description of `LIB$CREATE_VM_ZONE` (see the reference section of this manual).

The zone attributes are as follows:

- Allocation algorithms

The Run-Time Library heap management routines provide four algorithms to allocate and free memory, and to manage blocks of free memory. The algorithms are listed here. (See Section 5.5 for more details.)

- 1 The First Fit algorithm (`LIB$K_VM_FIRST_FIT`) maintains a linked list of free blocks, sorted in order of increasing memory address.
- 2 The Quick Fit algorithm (`LIB$K_VM_QUICK_FIT`) maintains a set of lookaside lists indexed by request size for request sizes in a specified range. For request sizes that are not in the specified range, a First Fit list of free blocks is maintained by the heap management routines.
- 3 The Frequent Sizes algorithm (`LIB$K_VM_FREQ_SIZES`) is similar to Quick Fit in that it maintains a set of lookaside lists for some block sizes. You specify the number of lists when you create the zone; the sizes associated with those lists are determined by the actual sizes of blocks that are freed.
- 4 The Fixed Size algorithm (`LIB$K_VM_FIXED`) maintains a single queue of free blocks.

- Boundary-tagged blocks

You can specify the use of boundary tags (`LIB$M_VM_BOUNDARY_TAGS`) with any of the algorithms that handle variable-sized blocks. The algorithms that handle variable-sized blocks are First Fit, Quick Fit, and Frequent Sizes.

If you specify boundary tags, `LIB$GET_VM` appends two additional longwords to each block that you allocate. `LIB$FREE_VM` uses these tags to speed up the process of merging adjacent free blocks on the First Fit free list. Using the standard First Fit insertion and merge, the execution time and number of page faults to free a block are proportional to the number of items on the list; freeing  $N$  blocks takes time proportional to  $N$  squared. When boundary tags are used, `LIB$FREE_VM` does not have to keep the free list in sorted order. This reduces the time and the number of page faults for freeing one block to a constant value that is independent of the number of free blocks. By using boundary tags you can improve execution time at the cost of some additional memory for the tags.

# Memory Allocation Routines

## 5.3 Zones

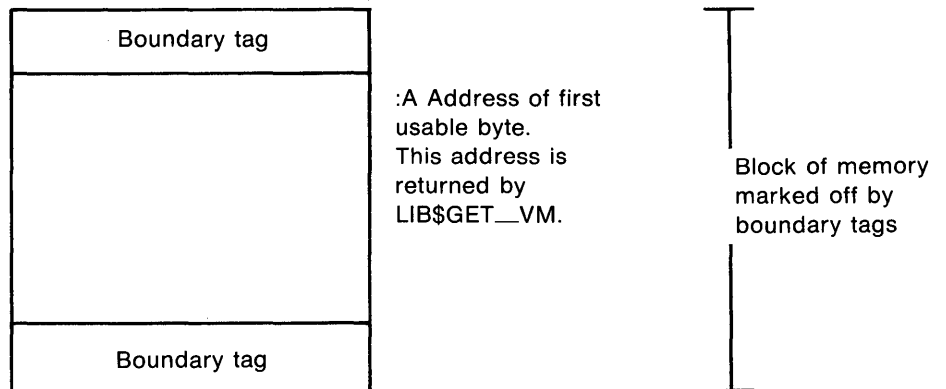
The use of boundary tags can have a significant effect on execution time if *all* of the following three conditions are present.

- 1 You are using the First Fit algorithm.
- 2 There are many calls to LIB\$FREE\_VM.
- 3 The free list is long.

Boundary tags will not improve execution time if you are using Quick Fit or Frequent Sizes and if all the blocks being freed use one of the lookaside lists. No merging or searching is done for free blocks on a lookaside list.

The boundary tags specify the length of each block that is allocated, so you do not need to specify the length of a tagged block when you free it. This reduces the bookkeeping that your program must perform. Figure 5-4 shows the placement of boundary tags.

**Figure 5-4 Boundary Tags**



ZK-4149-85

Boundary tags are not visible to the calling program. The request size you specify when calling LIB\$GET\_VM is the number of usable bytes your program needs. The address returned by LIB\$GET\_VM is the address of the first usable byte of the block, and this same address is used when you call LIB\$FREE\_VM.

- Area extension size

Pages of memory are allocated to a zone in contiguous groups called areas. By specifying area extension parameters for the zone, you can tailor the zone to achieve a satisfactory balance between locality, memory usage, and execution time for allocating pages. If you specify a large area size, you will improve locality for blocks in the zone, but you may waste a large amount of virtual memory. Pages may be allocated to an area of a zone, but the memory might never be used to satisfy a LIB\$GET\_VM allocation request. If you specify a small area extension size, you will reduce the number of pages used, but you may reduce locality and you will increase the amount of overhead for area control blocks.

# Memory Allocation Routines

## 5.3 Zones

You can specify two area extension size values: an initial size and an extend size. If you specify an initial area size, that number of pages will be allocated to the zone when you create the zone. If you do not specify an initial size, no pages are allocated until the first call to `LIB$GET_VM` that references the zone. When an allocation request cannot be satisfied by blocks from the free list or from memory in any of the areas owned by the zone, a new area is added to the zone. The size of this area is the maximum of the area extend size and the current request size. The extend size does not limit the size of blocks you can allocate. If you do not specify extend size when you create the zone, a default of 16 pages is used.

You should choose a few area extension sizes and use them throughout your program. It is also desirable to choose extension sizes that are multiples of each other. Memory for areas is allocated by calling `LIB$GET_VM_PAGE`. You should choose the area extension sizes in order to minimize fragmentation. DIGITAL-supplied software generally uses extension sizes that are a power of 2.

You should also consider the overhead for area control blocks when choosing the area extension parameters. Each area control block is 64 bytes long. Table 5-1 shows the overhead for various extension sizes.

**Table 5-1 Overhead for Area Control Blocks**

Area Size (in pages)	Overhead Percentage
1	12.5 %
2	6.3 %
4	3.1 %
16	0.8 %
128	0.1 %

You can also control the way in which zones are extended by using the `LIB$M_VM_EXTEND_AREA` attribute. This attribute specifies that when new pages are allocated for a zone, they should be appended to an existing area if the pages are adjacent to an existing area.

- **Block size**

The block size attribute specifies the number of bytes in the basic allocation quantum for the zone.

All allocation requests are rounded up to a multiple of the block size.

The block size must be a power of 2 in the range 8 to 512. Table 5-2 lists the possible block sizes that may be used.

# Memory Allocation Routines

## 5.3 Zones

**Table 5–2 Possible Values for the Block Size Attribute**

Power of 2	Actual Block Size
$2^3$	8
$2^4$	16
$2^5$	32
$2^6$	64
$2^7$	128
$2^8$	256
$2^9$	512

By adjusting the block size, you can control the effects of internal fragmentation and external fragmentation. Internal fragmentation occurs when the request size is rounded up and more bytes are allocated than are required to satisfy the request. External fragmentation occurs when there are many small blocks on the free list, but none of them is large enough to satisfy an allocation request.

If you do not specify a value for block size, a default of eight bytes is used.

- **Alignment**

The alignment attribute specifies the required address boundary alignment for each block allocated. The alignment value must be a power of 2 in the range 4 to 512.

The block size and alignment values are closely related. If you are not using boundary-tagged blocks, the larger value of block size and alignment controls both the block size and alignment. If you are using boundary-tagged blocks, you can minimize the overhead for the boundary tags by specifying a block size of 8 and an alignment of 4 (longword alignment). Note that the VAX interlocked queue instructions require quadword alignment, so you should not specify longword alignment for blocks that will be inserted on an interlocked queue.

If you do not specify an alignment value, a default of 8 is used (alignment on a quadword boundary).

- **Page limit**

You can specify the maximum number of 512-byte pages that can be allocated to the zone. If you do not specify a page limit, the only limit is the total process virtual address limit imposed by VMS process quotas and system parameters.

- **Fill on allocate**

If you do not specify the allocation-fill attribute, LIB\$GET\_VM does not initialize the contents of the blocks of memory that it supplies. The contents of the memory are unpredictable, and you must assign a value to each location your program uses.

In many applications, it is convenient to initialize every byte of dynamically allocated memory to the value 0. You can request that LIB\$GET\_VM do this initialization by specifying the allocation-fill attribute LIB\$M\_VM\_GET\_FILL0 when you create the zone.

# Memory Allocation Routines

## 5.3 Zones

If your program does not use the allocation-fill attribute, it may be very difficult to locate bugs where the program does not properly initialize dynamically allocated memory. As a debugging aid, you can request that LIB\$GET\_VM initialize every byte to FF (hexadecimal) by specifying the allocation-fill attribute LIB\$M\_VM\_GET\_FILL1 when you create the zone.

- Fill on free

In complex programs using heap storage, it can be very difficult to locate bugs where the program frees a block of memory but continues to make references to that block of memory. As a debugging aid, you can request that LIB\$FREE\_VM write bytes containing 0 or FF (hexadecimal) into each block of memory when it is freed; specify one of the attributes LIB\$M\_VM\_FREE\_FILLO or LIB\$M\_VM\_FREE\_FILL1.

---

### 5.3.2 The Default Zone

The Run-Time Library provides a default zone that is used if you do not specify a **zone-id** argument when you call LIB\$GET\_VM or LIB\$FREE\_VM. The default zone provides compatibility with earlier versions of LIB\$GET\_VM and LIB\$FREE\_VM, which did not support multiple zones.

Programs that do not place heavy demands on heap storage can simply use the default zone for all heap storage allocation. They do not need to call LIB\$CREATE\_VM\_ZONE and LIB\$DELETE\_VM\_ZONE, and they can omit the **zone-id** argument when calling LIB\$GET\_VM and LIB\$FREE\_VM. The **zone-id** for the default zone has the value zero.

The default zone has the values shown in Table 5-3.

**Table 5-3 Attribute Values for the Default Zone**

Attribute	Value
Algorithm	First Fit
Area extension size	128 pages
Block size	8 bytes
Alignment	Quadword boundary
Boundary tags	No boundary tags
Page limit	No page limit
Fill on allocate	No fill on allocate
Fill on free	No fill on free

---

### 5.3.3 Zone Identification

A zone is a logically independent memory pool or subheap. You can create zones by calling LIB\$CREATE\_VM\_ZONE or LIB\$CREATE\_USER\_VM\_ZONE. These routines return as an output argument a unique 32-bit zone identifier (**zone-id**) which is used in subsequent routine calls where a zone identification is needed.

# Memory Allocation Routines

## 5.3 Zones

You can specify **zone-id** as an optional argument when you call `LIB$GET_VM` to allocate a block of memory. If you do specify **zone-id** when you allocate memory, you must specify the same **zone-id** value when you call `LIB$FREE_VM` to free the memory. `LIB$FREE_VM` will return an error status if you do not provide the correct **zone-id**.

Modular routines that allocate and free heap storage must use zone identifications in a consistent fashion. There are several approaches you can use in designing a set of modular routines to ensure consistency in using zone identifications:

- Each routine that allocates or frees heap storage has a **zone-id** argument so the caller can specify the zone to be used.
- The modular routine package provides `ALLOCATE` and `FREE` routines for each type of dynamically allocated object. These routines keep track of zone identifications in an implicit argument, in static storage, or in the dynamically allocated objects. The caller need not be concerned with the details of zone identifications.
- By convention, the set of modular routines could do all allocate and free operations in the default zone.

The zone identifier for the default zone has the value 0 (see Section 5.3.2 for more information on the default zone). You can allocate and free blocks of memory in the default zone by specifying a **zone-id** of 0 or by omitting the **zone-id** argument when you call `LIB$GET_VM` and `LIB$FREE_VM`. You cannot use `LIB$DELETE_VM_ZONE` or `LIB$RESET_VM_ZONE` on the default zone; these routines return an error status if the **zone-id** is 0.

---

### 5.3.4 Creating a Zone

`LIB$CREATE_VM_ZONE` creates a new zone and sets zone attributes according to arguments that you supply. `LIB$CREATE_VM_ZONE` returns a zone-id value for the new zone that you use in subsequent calls to `LIB$GET_VM`, `LIB$FREE_VM`, and `LIB$DELETE_VM_ZONE`.

---

### 5.3.5 Deleting a Zone

`LIB$DELETE_VM_ZONE` deletes a zone and returns all pages owned by the zone to the processwide page pool managed by `LIB$GET_VM_PAGE`. Your program must not do any further operations on the zone after you call `LIB$DELETE_VM_ZONE`.

It takes less execution time to free memory in a single operation by calling `LIB$DELETE_VM_ZONE` than to individually account for and free every block of memory that was allocated by calling `LIB$GET_VM`. Of course, you must be sure that your program is no longer using the zone or any of the memory in the zone before you call `LIB$DELETE_VM_ZONE`.

If you have specified deallocation filling, `LIB$DELETE_VM_ZONE` will fill all of the allocated blocks that are freed.

# Memory Allocation Routines

## 5.3 Zones

---

### 5.3.6 Resetting a Zone

`LIB$RESET_VM_ZONE` frees all the blocks of memory that were previously allocated from the zone. The memory becomes available to satisfy further allocation requests for the zone; the memory is not returned to the processwide page pool managed by `LIB$GET_VM_PAGE`. Your program can continue to use the zone after you call `LIB$RESET_VM_ZONE`.

It takes less execution time to free memory in a single operation by calling `LIB$RESET_VM_ZONE` than to individually account for and free every block of memory that was allocated by calling `LIB$GET_VM`. Of course, you must be sure that your program is no longer using any of the memory in the zone before you call `LIB$RESET_VM_ZONE`.

If you have specified deallocation filling, `LIB$RESET_VM_ZONE` will fill all of the allocated blocks that are freed.

Since `LIB$RESET_VM_ZONE` does not return any pages to the processwide page pool, you should reset a zone only if you expect to reallocate almost all of the memory that is currently owned by the zone. If the next cycle of reallocation might use much less memory, it is better to delete the zone (`LIB$DELETE_VM_ZONE`) and create it (`LIB$CREATE_VM_ZONE`) again.

---

## 5.4 Allocating and Freeing Blocks

The Run-Time Library heap management routines `LIB$GET_VM` and `LIB$FREE_VM` provide the mechanism for allocating and freeing blocks of memory.

`LIB$GET_VM` and `LIB$FREE_VM` are fully reentrant, so they can be called by code running at AST level or in an Ada multitasking environment. Several threads of execution may be simultaneously allocating or freeing memory in the same zone or in different zones.

All memory allocated by `LIB$GET_VM` has user-mode read-write access, even if the call to `LIB$GET_VM` is made from a more privileged access mode.

The rules for using `LIB$GET_VM` and `LIB$FREE_VM` are as follows:

- Any memory you free by calling `LIB$FREE_VM` must have been allocated by a previous call to `LIB$GET_VM`. You cannot allocate memory by calling `$EXPREG` or `$CRETVA` and then free it using `LIB$FREE_VM`.
- When you free a block of memory by calling `LIB$FREE_VM`, you must use the same **zone-id** value as when you called `LIB$GET_VM` to allocate the block. If the block was allocated from the default zone, you must either specify a **zone-id** of zero or omit the **zone-id** argument when you call `LIB$FREE_VM`.
- You cannot free part of a block that was allocated by a call to `LIB$GET_VM`; the whole block must be freed by a single call to `LIB$FREE_VM`.
- You cannot combine contiguous blocks of memory that were allocated by several calls to `LIB$GET_VM` into one larger block that is freed by a single call to `LIB$FREE_VM`.

# Memory Allocation Routines

## 5.4 Allocating and Freeing Blocks

- All memory allocated by LIB\$GET\_VM is aligned according to the alignment attribute for the zone; all memory freed by LIB\$FREE\_VM must have the correct alignment for the zone. An error status is returned if you attempt to free a block that is not aligned properly.

## 5.5 Allocation Algorithms

The Run-Time Library heap management routines provide four algorithms that are used to allocate and free memory and that are used to manage blocks of free memory.

**Table 5–4 Allocation Algorithms**

Code	Symbol	Description
1	LIB\$_VM_FIRST_FIT	First Fit
2	LIB\$_VM_QUICK_FIT	Quick Fit (maintains lookaside list)
3	LIB\$_VM_FREQ_SIZES	Frequent Sizes (maintains lookaside list)
4	LIB\$_VM_FIXED	Fixed Size Blocks

The Quick Fit and Frequent Sizes algorithms use lookaside lists to speed up allocation and freeing for certain request sizes. A lookaside list is the software analog of a hardware cache. It takes less time to allocate or free a block on a lookaside list.

For each of the algorithms, LIB\$GET\_VM performs one or more of the following operations:

- Tries to allocate a block from an appropriate lookaside list.
- Scans the list of areas owned by the zone. For each area, tries to allocate a block from the free list; then tries to allocate a block from the block of unallocated memory at the end of the area.
- Adds a new area to the zone and allocates the block from that area.

For each of the algorithms, LIB\$FREE\_VM performs one or more of the following operations:

- Places the block on a lookaside list associated with the zone if there is an appropriate list.
- Locates the area that contains the block. If the zone has boundary tags, the tags encode the area; otherwise, it scans the list of areas owned by the zone to find the correct area.
- Inserts the block on the area free list and checks for merges with adjacent free blocks.

If the zone has boundary tags, LIB\$FREE\_VM checks the tags of adjacent blocks; if no merge occurs, it inserts the block at the tail of the free list.

If the zone does not have boundary tags, LIB\$FREE\_VM scans the sorted free list to find the correct insertion point. It also checks the preceding and following blocks for merges.



# Memory Allocation Routines

## 5.5 Allocation Algorithms

---

### 5.5.1 The First Fit Algorithm

The First Fit algorithm (LIB\$K\_VM\_FIRST\_FIT) maintains a linked list of free blocks. If the zone does not have boundary tags, the free list is kept sorted in order of increasing memory address. An allocation request is satisfied by the first block on the free list that is large enough; if the first free block is larger than the request size, it is split and the fragment is kept on the free list. When a block is freed, it is inserted in the free list at the appropriate point; adjacent free blocks are merged to form larger free blocks.

---

### 5.5.2 The Quick Fit Algorithm

The Quick Fit algorithm (LIB\$K\_VM\_QUICK\_FIT) maintains a set of lookaside lists indexed by request size for request sizes in a specified range. For request sizes that are not in the specified range, a First Fit list of free blocks is maintained. An allocation request is satisfied by removing a block from the appropriate lookaside list; if the lookaside list is empty, a First Fit allocation is done. When a block is freed, it is placed on a lookaside list or the First Fit list according to its size.

Free blocks that are placed on a lookaside list are neither merged with adjacent free blocks nor split to satisfy a request for a smaller block.

---

### 5.5.3 The Frequent Sizes Algorithm

The Frequent Sizes algorithm (LIB\$K\_VM\_FREQ\_SIZES) is similar to the Quick Fit algorithm in that it maintains a set of lookaside lists for some block sizes. You specify the number of lookaside lists when you create the zone; the sizes associated with those lists are determined by the actual sizes of blocks that are freed. An allocation request is satisfied by searching the lookaside lists for a matching size; if no match is found, a First Fit allocation is done. When a block is freed, the block is placed on a lookaside list with matching size, on an empty lookaside list, or on the First Fit list if no lookaside list is available. Comparable to the Quick Fit algorithm, free blocks on lookaside lists are not merged or split.

---

### 5.5.4 The Fixed Size Algorithm

The Fixed Size algorithm (LIB\$K\_VM\_FIXED) maintains a single queue of free blocks. There is no First Fit free list, and no splitting or merging of blocks occurs.

---

## 5.6 User-Defined Zones

When you create a zone by calling LIB\$CREATE\_VM\_ZONE, you must select an allocation algorithm from the fixed set provided by the Run-Time Library. You can tailor the characteristics of the zone by specifying various zone attributes. User-defined zones provide additional flexibility and control by letting you supply routines for the allocation and deallocation algorithms.

# Memory Allocation Routines

## 5.6 User-Defined Zones

You create a user-defined zone by calling `LIB$CREATE_USER_VM_ZONE`. Instead of supplying values for a fixed set of zone attributes, you provide routines that perform the following operations for the zone:

- Allocate a block of memory
- Free a block of memory
- Reset the zone
- Delete the zone

Each time that one of the Run-Time Library heap management routines (`LIB$GET_VM`, `LIB$FREE_VM`, `LIB$RESET_VM_ZONE`, `LIB$DELETE_VM_ZONE`) is called to perform an operation on a user-defined zone, the corresponding routine that you specified is called to perform the actual operation. It is not necessary to make any changes in the calling program to use user-defined zones; their use is transparent.

You do not need to provide routines for all four of the operations listed above if you know that your program will not perform certain operations. If you omit some of the operations and your program attempts to use them, an error status will be returned.

Applications of user-defined zones include the following:

- You can provide your own specialized allocation algorithms. These algorithms can in turn invoke `LIB$GET_VM`, `LIB$GET_VM_PAGE`, `$EXPREG`, or other VMS system services.
- You can use a user-defined zone to monitor memory allocation operations. Example 5-1 shows a monitoring program that prints a record of each call to allocate or free memory in a zone.

### Example 5-1 Monitoring Heap Operations with a User-Defined Zone

---

```
C+
C This is the main program that creates a zone and exercises it.
C
C Note that the main program simply calls LIB$GET_VM and LIB$FREE_VM.
C It contains no special coding for user-defined zones.
C-
```

```
PROGRAM MAIN
IMPLICIT INTEGER(A-Z)

CALL MAKE_ZONE(ZONE)

CALL LIB$GET_VM(10, I1, ZONE)
CALL LIB$GET_VM(20, I2, ZONE)
CALL LIB$FREE_VM(10, I1, ZONE)
CALL LIB$RESET_VM_ZONE(ZONE)
CALL LIB$DELETE_VM_ZONE(ZONE)
END
```

```
C+
C This is the subroutine that creates a user-defined zone for monitoring.
C Each GET, FREE, or RESET prints a line of output on the terminal.
C Errors are signaled.
C-
```

---

Example 5-1 Cont'd. on next page

# Memory Allocation Routines

## 5.6 User-Defined Zones

### Example 5–1 (Cont.) Monitoring Heap Operations with a User-Defined Zone

---

```
        SUBROUTINE MAKE_ZONE(ZONE)
        IMPLICIT INTEGER (A-Z)
        EXTERNAL GET_RTN, FREE_RTN, RESET_RTN, LIB$DELETE_VM_ZONE

C+
C Create the primary zone. The primary zone supports
C the actual allocation and freeing of memory.
C-

        STATUS = LIB$CREATE_VM_ZONE(REAL_ZONE)
        IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C+
C Create a user-defined zone that monitors operations on REAL_ZONE.
C-

        STATUS = LIB$CREATE_USER_VM_ZONE(USER_ZONE, REAL_ZONE,
        1      GET_RTN,
        1      FREE_RTN,
        1      RESET_RTN,
        1      LIB$DELETE_VM_ZONE)
        IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C+
C Return the zone-id of the user-defined zone to the caller to use.
C-

        ZONE = USER_ZONE
        END

C+
C GET routine for user-defined zone.
C-

        FUNCTION GET_RTN(SIZE, ADDR, ZONE)
        IMPLICIT INTEGER(A-Z)

        STATUS = LIB$GET_VM(SIZE, ADDR, ZONE)
        IF (.NOT. STATUS) THEN
            CALL LIB$SIGNAL(%VAL(STATUS))
        ELSE
            TYPE 10, SIZE, ADDR
10          FORMAT(' Allocated ',I4,' bytes at ',Z8)
        ENDIF
        GET_RTN = STATUS
        END
```

---

Example 5–1 Cont'd. on next page

# Memory Allocation Routines

## 5.6 User-Defined Zones

### Example 5-1 (Cont.) Monitoring Heap Operations with a User-Defined Zone

---

```
C+
C FREE routine for user-defined zone.
C-

        FUNCTION FREE_RTN(SIZE, ADDR, ZONE)
        IMPLICIT INTEGER(A-Z)

        STATUS = LIB$FREE_VM(SIZE, ADDR, ZONE)

        IF (.NOT. STATUS) THEN
            CALL LIB$SIGNAL(%VAL(STATUS))
        ELSE
            TYPE 20, SIZE, ADDR
20          FORMAT(' Freed ',I4,' bytes at ',Z8)
        ENDIF
        FREE_RTN = STATUS
        END

C+
C RESET routine for user-defined zone.
C-

        FUNCTION RESET_RTN(ZONE)
        IMPLICIT INTEGER(A-Z)

        STATUS = LIB$RESET_VM_ZONE(ZONE)
        IF (.NOT. STATUS) THEN
            CALL LIB$SIGNAL(%VAL(STATUS))
        ELSE
            TYPE 30, ZONE
30          FORMAT(' Reset zone at ', Z8)
        ENDIF
        RESET_RTN = STATUS
        END
```

---

## 5.7 Interactions with Other Run-Time Library Routines

Section 5.1 describes a three-level hierarchy of memory allocation routines consisting of the following:

- 1 VMS memory management system services
- 2 Run-Time Library page management routines LIB\$GET\_VM\_PAGE and LIB\$FREE\_VM\_PAGE
- 3 Run-Time Library heap management routines LIB\$GET\_VM and LIB\$FREE\_VM

The Run-Time Library and various VAX programming languages provide another level of more specialized allocation routines.

- The Run-Time Library dynamic string package provides a set of routines for allocating and freeing dynamic strings. The set of routines includes the following:

```
LIB$SGET1_DD, LIB$SFREE1_DD
LIB$SFREEN_DD
STR$GET1_DX, STR$FREE1_DX
```

# Memory Allocation Routines

## 5.7 Interactions with Other Run-Time Library Routines

- VAX Ada provides allocators and the UNCHECKED\_DEALLOCATION package for allocating and freeing memory.
- VAX PASCAL provides the NEW and DISPOSE routines for allocating and freeing memory.
- VAX PL/I provides ALLOCATE and FREE statements for allocating and freeing memory.

A program containing routines written in several VAX languages may use a number of these facilities at the same time. This does not cause any problems or impose any restrictions on the user since all of these are layered on the Run-Time Library heap management routines.

**Note:** To ensure correct operation, memory that is allocated by one of the higher-level allocators in the preceding list can only be freed by using the corresponding deallocation routine. That is, memory allocated by PASCAL NEW must be freed by calling PASCAL DISPOSE, and a dynamic string can be freed only by calling one of the string package deallocation routines.

---

## 5.8 Interactions with VMS System Services

The Run-Time Library page management and heap management routines are implemented as layers built on the VMS memory management system services. In general, modular routines should use the Run-Time Library routines rather than directly call VMS memory management system services. However, there are some situations where you must use both. This section describes relationships between the Run-Time Library and VMS memory management. See the *VMS System Services Reference Manual* for descriptions of the memory management system services.

You can use the Expand Region system service (\$EXPREG) to create pages of virtual memory in the program region (P0 space) for your process. VMS keeps track of the first free page address at the end of P0 space, and it updates this free page address whenever you call \$EXPREG or \$CRETVA. The LIB\$GET\_VM\_PAGE routine calls \$EXPREG to create pages, so there will be no conflicting address assignments when you call \$EXPREG directly.

You should avoid using the Create Virtual Address Space system service (\$CRETVA), because you must specify the range of virtual addresses when it is called. If the address range you specify contains pages that already exist, \$CRETVA deletes those pages and re-creates them as demand-zero pages. It may be difficult to avoid conflicting address assignments if you use Run-Time Library routines and \$CRETVA.

You must not use the Contract Region system service (\$CNTREG) because other routines or the VAX Record Management Services (RMS) may have allocated pages at the end of the program region.

You can change the protection on pages your program has allocated by calling the Set Protection system service (\$SETPRT). All pages allocated by LIB\$GET\_VM\_PAGE have user-mode read-write access. If you change protection on pages allocated by LIB\$GET\_VM\_PAGE, you must reset the protection to user-mode read-write before calling LIB\$FREE\_VM\_PAGE to free the pages.

# Memory Allocation Routines

## 5.8 Interactions with VMS System Services

You can use the Create and Map Section system service (`$CRMPSC`) to map a file into your virtual address space. To map a file, you provide a range of virtual addresses for the file. One way to do this is to specify the Expand Region option (`SEC$M_EXPREG`) when you call `$CRMPSC`. This method assigns addresses at the end of P0 space, similar to the `$EXPREG` system service. Alternatively, you can provide a specific range of virtual addresses when you call `$CRMPSC`; this is similar to allocating pages by calling `$CRETVA`. If you assign a specific range of addresses, you must avoid conflicts with other routines. One way to do this is to allocate memory by calling `LIB$GET_VM_PAGE`, then use that memory to map the file. The complete sequence of steps is as follows:

- 1 Call `LIB$GET_VM_PAGE` to allocate a contiguous group of (n+1) pages. The first n pages will be used to map the file; the last page serves as a guard page.
- 2 Call `$CRMPSC` using the first n pages to map the file into your process address space.
- 3 Process the file.
- 4 Call `$DELTVA` to delete the first n pages and unmap the file.
- 5 Call `$CRETVA` to recreate the n pages of virtual address space as demand-zero pages.
- 6 Call `LIB$FREE_VM_PAGE` to free (n+1) pages of memory and return them to the processwide page pool.

The sequence is satisfactory when mapping small files of a few hundred pages, but it has severe limitations when mapping very large files. As discussed in Section 5.2, you should not use `LIB$GET_VM_PAGE` to allocate very large groups of contiguous pages (over 1000 contiguous pages in a single request). Also, when you allocate memory by calling `LIB$GET_VM_PAGE` (and thus `$EXPREG`), the pages are charged against your process page file quota. Your page file quota is not charged if you call `$CRMPSC` with the `SEC$M_EXPREG` option.

You can process very large files using `$CRMPSC` by first providing a pool of pages that is sufficient for your program and then using `$CRMPSC` and `$DELTVA` to map and unmap the file. Use `LIB$SHOW_VM` to obtain an estimate of how much dynamically allocated memory your program requires; round this number up and allow for increased memory usage in the future. You can then use the memory estimate as follows:

- 1 At the beginning of your program, include code to call `LIB$GET_VM_PAGE` and allocate the estimated number of pages. You should not request a large number of pages in one call to `LIB$GET_VM_PAGE`, because this would require contiguous allocation of the pages.
- 2 Call `LIB$FREE_VM_PAGE` to free all the pages allocated in Step 1; this establishes a pool of free pages for your program.
- 3 Open files that your program needs; note that RMS may allocate buffers in P0 space.

# Memory Allocation Routines

## 5.8 Interactions with VMS System Services

- 4 Call \$CRMPSC specifying SEC\$M\_EXPREG to map the file into your process address space at the end of P0 space.
- 5 Process the file.
- 6 Call \$DELTVA specifying the address range to release the file. If no additional pages were created after you mapped the file, \$DELTVA will contract your address space. Your program can repeat the process of mapping a file without continually expanding its address space.

# 6

## Debugging Programs That Use Virtual Memory Zones

---

This chapter discusses some methods and aids for debugging programs that use virtual memory zones. It is important to note that this information is implementation-dependent and may change at any time.

The following list offers some suggestions for discovering and tracking down problems with memory zone usage.

- Run the program with both free-fill-zero and free-fill-one set. The results from both executions of the program should be the same. If the results differ, this could mean that you are referencing a zone that is already deallocated. It could also mean that after deallocating a zone, you created a new zone at the same location, so that you now have two pointers pointing to the same zone.
- Call `LIB$FIND_VM_ZONE` at image termination. If a virtual memory zone is not deleted, `LIB$FIND_VM_ZONE` will return its zone identifier.
- Use `LIB$SHOW_VM_ZONE` and `LIB$VERIFY_VM_ZONE` to print zone information and check for errors in the internal data structures. `LIB$SHOW_VM_ZONE` allows you to determine whether any linkage pointers for the virtual memory zones are corrupted. `LIB$VERIFY_VM_ZONE` allows you to request verification of the contents of the free blocks, so that if you call `LIB$VERIFY_VM_ZONE` with free-fill set, you can determine whether you are writing to any deallocated zones.
- For zones created with either the Fixed Size, Quick Fit, or Frequent Size algorithms, some types of errors cannot be detected. For example, in a zone implementing the fixed size algorithm (or in a Quick Fit or Frequent Size algorithm when the block is cached on a lookaside list), freeing a block more than once returns `SS$_NORMAL` but the internal data structures are invalid. In this case, you should change the algorithm to First Fit. The First Fit algorithm checks to see if you are freeing a block that is already on the free list, and if so, returns the error `LIB$_BADBLOADR`.





# 7

---

## Image Initialization and Termination

This chapter describes the system declaration mechanism, including `LIB$INITIALIZE`, which performs calls to any initialization routine declared by the user. This mechanism is available to the Run-Time Library so that user routines that require special initialization can be added to the library. However, use of `LIB$INITIALIZE` is discouraged and should be used only when no other method is suitable.

In most cases, both user and library routines are self-initializing. This means that they can process information with no special action required by the calling program. Initialization is automatic because 1) the routine's statically allocated data storage is initialized at compile or link time, or 2) a statically allocated flag is tested and set on each call so initialization occurs only on the first call.

Any special initialization—such as a call to other routines or to system services—can be performed on the first call before the main program is initialized. For example, you can establish a new environment to alter the way errors are handled or the way messages are printed.

Such special initialization is required only rarely; however, you do not need to require the caller of the routine to make an explicit initialization call. The Run-Time Library provides a system declaration mechanism that performs all such initialization calls before the main program is called. Special initialization is thus invisible to later callers of the routine.

---

### 7.1 Image Initialization

Before the main program or main routine is called, a number of system initialization routines are called as specified by a 1-, 2-, or 3-longword initialization list set up by the linker. This initialization list consists of the addresses of the debugger (if present), the `LIB$INITIALIZE` routine (if present), and the entry point of the main program or main routine, in that order. The following initialization steps take place:

- 1 The image activator maps the user program into the address space of the process and sets up useful information such as the program name. Then it starts up the command interpreter.
- 2 The command interpreter sets up an argument list and calls the next routine in the initialization list (debugger, `LIB$INITIALIZE`, main program, or main routine).
- 3 The debugger, if present, initializes itself and calls the next routine in the initialization list (`LIB$INITIALIZE`, main program, or main routine).
- 4 `LIB$INITIALIZE`, if present, is a library routine that calls each library and user initialization routine declared using the system `LIB$INITIALIZE` mechanism. Then it calls the main program or main routine.

# Image Initialization and Termination

## 7.1 Image Initialization

- 5 The main program or main routine executes and, at the user's discretion, accesses its argument list to scan the command or to obtain information about the image. The main program or main routine can then call other routines.
- 6 Eventually, the main program or main routine terminates by executing a return instruction (RET) with R0 set to a standard completion code to indicate success or failure, where bit 0 equals 1 for success or 0 for failure.
- 7 The completion code is returned to LIB\$INITIALIZE (if present), the debugger (if present), and finally to the command interpreter, which issues a \$EXIT system service with the completion status as an argument. Any declared exit handlers are called at this point.

**Note:** Main programs should not call the \$EXIT system service directly. If they do, other programs cannot call them as routines.

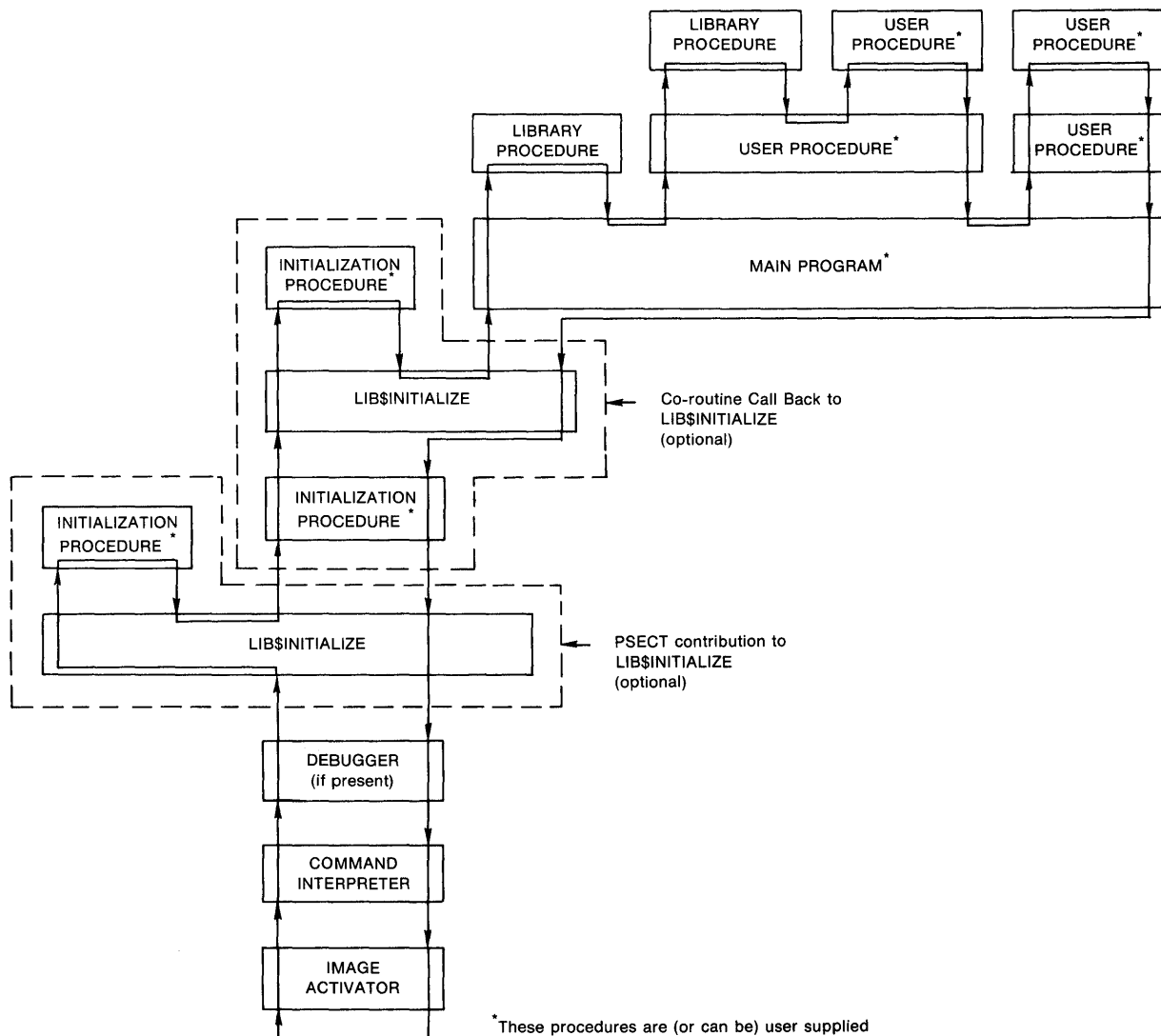
Figure 7-1 illustrates the sequence of calls and returns in a typical image initialization. Each box is a routine activation as represented on the image stack. The top of the stack is at the top of the figure. Each upward arrow represents the result of a CALLS or CALLG instruction that creates a routine activation on the stack to which control is being transferred. Each downward arrow represents the result of a RET (return) instruction. A RET instruction removes the routine activation from the stack and causes control to be transferred downward to the next box.

A user program can alter the image initialization sequence by making a PSECT contribution to PSECT LIB\$INITIALIZE and declaring EXTERNAL LIB\$INITIALIZE. This adds the optional initialization steps shown in Figure 7-1 labeled "PSECT contribution to LIB\$INITIALIZE." (A PSECT is a portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.) If the initialization routine also performs a coroutine call back to LIB\$INITIALIZE, the optional steps labeled "Coroutine call back to LIB\$INITIALIZE" shown in Figure 7-1 are added to the image initialization sequence.

# Image Initialization and Termination

## 7.1 Image Initialization

Figure 7-1 Sequence of Events During Image Initialization



ZK-1977-84

## 7.2 Initialization Argument List

The following argument list is passed from the command interpreter, the debugger, or LIB\$INITIALIZE to the main program. This argument list is the same for each routine activation.

(start ,cli-coroutine [,image-info])

The **start** argument is the address of the entry in the initialization vector that is used to perform the call.

# Image Initialization and Termination

## 7.2 Initialization Argument List

The **cli-routine** argument is the address of a command interpreter coroutine to obtain command arguments.

The **image-info** argument is useful image information such as the program name.

The debugger or LIB\$INITIALIZE, or both, can call the next routine in the initialization chain using the following coding sequence:

```
.  
.  
.  
ADDL    #4, 4(AP)      ; Step to next initialization list entry  
MOVL    @4(AP), RO    ; RO = next address to call  
CALLG   (AP), (RO)    ; Call next initialization routine  
.  
.  
.
```

This coding sequence modifies the contents of an argument list entry. Thus, the sequence does not follow the VAX Procedure Calling and Condition Handling Standard. However, the argument list can be expanded in the future without requiring any change either to the debugger or to LIB\$INITIALIZE.

---

## 7.3 Declaring Initialization Routines

Any library or user program module can declare an initialization routine. This routine will be called when the image is started. The declaration is made by making a contribution to PSECT LIB\$INITIALIZE, which contains a list of routine entry point addresses to be called before the main program or main routine is called.

The following MACRO example declares an initialization routine by placing the routine entry address INIT\_PROC in the list:

```
.EXTRN LIB$INITIALIZE      ; cause library initialization  
                          ; dispatcher to be loaded  
.PSECT LIB$INITIALIZE, NOPIC, USR, CON, REL, GBL, NOSHR, NOEXE, RD, NOWRT, LONG  
.LONG INIT_PROC          ; contribute entry point address of  
                          ; initialization routine.  
.PSECT ...
```

The .EXTRN declaration links the initialization routine dispatcher, LIB\$INITIALIZE, into your program's image. The reference contains a definition of the special global symbol LIB\$INITIALIZE, which is the routine entry point address of the dispatcher. The linker stores the value of this special global symbol in the initialization list along with the starting address of the debugger and the main program. The GBL specification ensures that the PSECT LIB\$INITIALIZE contribution is not affected by any clustering performed by the linker.

# Image Initialization and Termination

## 7.4 Dispatching to Initialization Routines

---

### 7.4 Dispatching to Initialization Routines

The LIB\$INITIALIZE dispatcher calls each initialization routine in the list with the following argument list.

```
CALL init-proc (init-coroutine ,cli-coroutine [, image-info])
```

The **init-coroutine** argument is the address of a library coroutine to be called to effect a coroutine linkage with LIB\$INITIALIZE.

The **cli-coroutine** is the address of a command interpreter coroutine used to obtain command arguments.

The **image-info** argument is useful image information such as the program name.

---

### 7.5 Initialization Routine Options

An initialization routine has a number of options. It can be used to do the following:

- Set up an exit handler by calling the Declare Exit Handler (\$DCLEXH) system service, although exit handlers are generally set up by using a statically allocated first-time flag.
- Initialize statically allocated storage, although this is preferably done at image activation time using compile-time and link-time data initialization declarations or using a first-time call flag in its statically allocated storage.
- Call the initialization dispatcher (instead of returning to it) by calling the **init-coroutine** argument. This achieves a coroutine link. Control will return to the initialization routine when the main program returns control. Then, the initialization routine should also return control to pass back the completion code returned by the main program (to the debugger or command interpreter, or both).
- Establish a condition handler in the current frame before performing the previous step. This will leave the initialization routine condition handler on the image stack for the duration of the image execution. This occurs after the command interpreter sets up the catch-all stack frame handler, and after the debugger sets up its stack frame handler. Thus, the initialization routine handler can override either of these handlers since it will receive signals before they do.

---

### 7.6 An Example

The following MACRO code fragment shows how an initialization routine does the following:

- Establishes a handler
- Calls the **init-coroutine** routine, so that the coroutine calls the initialization dispatcher
- Gains control after the main program returns
- Returns to the normal exit processing

# Image Initialization and Termination

## 7.6 An Example

```
INIT_PROC:
    .WORD ^M<>                ; no registers used
    MOVAL HANDLER, (FP)        ; establish handler
    ...                        ; perform any other initialization
    CALLG (AP), @INIT_CO_ROUTINE(AP)
10$:
    ...                        ; continue initialization which
                                ; then calls main program or
                                ; routine.
    ...                        ; Return here when main program
                                ; returns with RO = completion
    RET                        ; status return to normal exit
                                ; processing with RO = completion
                                ; status

HANDLER:
    .WORD ^M<...>            ; condition handler
    ...                        ; register mask
    ...                        ; handle condition
                                ; could unwind to 10$
    MOVL #..., RO            ; Set completion status with a
                                ; condition value
    RET                        ; resignal or continue depending
                                ; on RO being SS$_RESIGNAL or
                                ; SS$_CONTINUE.
```

---

## 7.7 Image Termination

Main programs and main routines terminate by executing a return instruction (RET). This returns control to the caller, which may have been LIB\$INITIALIZE, the debugger, or the command interpreter. The completion code, SS\$\_NORMAL, which has the value 1, should be used to indicate normal successful completion.

Any other condition value can be used to indicate success or failure. The condition value is used as the parameter to the exit (\$EXIT) system service by the command interpreter. If the severity field (STS\$V\_SEVERITY) is SEVERE or ERROR, the continuation of a batch job or command procedure is affected.

You should not call the \$EXIT system service directly from a main program. This allows the main programs to be more like ordinary modular routines and hence usable by other programmers as callable routines.

# 8

## Cross-Reference Routines

The cross-reference routines are contained in a separate, shareable image capable of creating a cross-reference analysis of symbols. They accept cross-reference data, summarize it, and format it for output. Two facilities that use the cross-reference routines are the VMS Linker and the MACRO assembler. They are sufficiently general, however, to be used by any native-mode utility.

Table 8-1 lists the entry points and functions of the cross-reference routines.

**Table 8-1 Cross-Reference Routines**

Entry Point	Function
LIB\$CRF_INS_KEY	Insert key information
LIB\$CRF_INS_REF	Insert reference information
LIB\$CRF_OUTPUT	Summarize and format cross-reference information

ZK-4259-85

The interface to the cross-reference routines is by way of a set of control blocks, format definition tables, and a set of callable entry points. Macros are provided for assembly language and BLISS initialization of the control blocks and format definition tables.

### 8.1 Using the Cross-Reference Routines

Using the cross-reference routines involves the following steps:

- 1 Define a table of control information, using the \$CRFCTLTABLE macro.
- 2 Define each field of the output line, using the \$CRFFIELD macro.
- 3 Specify the end of each set of macros that define a field in the output line, using the \$CRFFIELDEND macro.
- 4 Provide data by calling one of the two following cross-reference entry points:
  - LIB\$CRF\_INS\_KEY inserts an entry for the specified key in the specified symbol table.
  - LIB\$CRF\_INS\_REF inserts a reference to a key in the specified symbol table.
- 5 Call LIB\$CRF\_OUTPUT, the cross-reference output routine, to summarize and format the data.



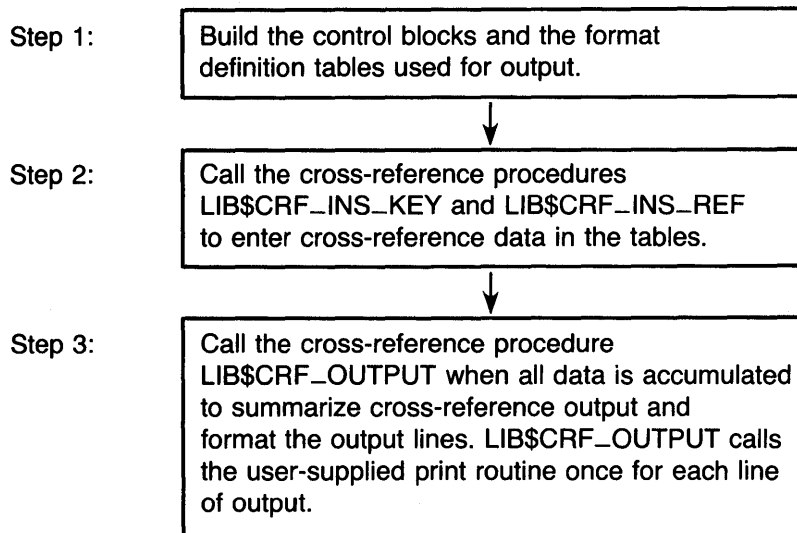
# Cross-Reference Routines

## 8.1 Using the Cross-Reference Routines

- 6 Supply a routine that the output routine calls to print each line in the output file. Because you supply this routine, you can control the number of lines per page and the header lines.

Figure 8-1 illustrates the steps required in using the cross-reference routines.

**Figure 8-1 Using Cross-Reference Routines**



ZK-1970-84

---

The Run-Time Library provides three macros to initialize the data structures used by the cross-reference routines:

- 1 `$CRFCTLTABLE` defines a table of control information.
- 2 `$CRFFIELD` defines each field of the output format definition table. Multiple `$CRFFIELD` macro instructions can be issued in defining one particular field.
- 3 `$CRFFIELDEND` ends a set of `$CRFFIELD` macro instructions (a format table).

---

## 8.2 `$CRFCTLTABLE` Macro

`$CRFCTLTABLE` initializes a cross-reference control table. Your program must issue one `$CRFCTLTABLE` macro for each cross-reference table you build. You can accumulate information for more than one cross-reference table at a time. For this reason, you must define a table for each set of cross-references, and include the address of that table each time you call a cross-reference routine to insert data.

# Cross-Reference Routines

## 8.2 \$CRFCTLTABLE Macro

The \$CRFCTLTABLE macro instruction has the following format:

```
label: $CRFCTLTABLE keytype, output, error, memexp, key1table,  
                    key2table, val1table, val2table,  
                    ref1table, ref2table
```

### label

The address of the control table. You must specify a control table address in all calls to the cross-reference routines.

### keytype

The type of key to enter into the table. The following key types are defined:

ASCIC	Keys are counted ASCII strings, with a maximum of 31 characters (symbol name).
BIN_U32	Keys are 32-bit unsigned binary values. The binary-to-ASCII conversion is done by \$FAO using the format string for the KEY1 field.

### output

The address of the routine that you supply to print a formatted output line. The output line is passed to the output routine by descriptor.

### error

The address of an error routine to execute if the called cross-reference routine encounters an error. The error code (longword) is passed to the error routine by value. In other words, it is a copy of the constant on the stack. A value of zero indicates that no error routine is supplied.

### memexp

The number of pages by which to expand region when needed. The default is 50.

### key1 table

The address of the field descriptor table for the KEY1 field. A value of zero indicates that the field is not to be included in the output line.

The remaining arguments provide the address of the field descriptor tables for the KEY2, VAL1, VAL2, REF1, and REF2 fields, respectively, of the output line. You can use these argument names as keywords in the macros. For example, you can use KEYTYPE as a keyword when issuing the \$CRFCTLTABLE macro.

---

## 8.3 \$CRFFIELD Macro

For each field in the output line, you must issue a \$CRFFIELD instruction to identify the field, supply an \$FAO command string to control the printing of the field, and provide flag information. See the program example and the description of \$FAO (formatted ASCII output) in the *VMS System Services Reference Manual*. The \$CRFFIELD macro has the following format:

```
label: $CRFFIELD bit_mask, fao_string, field_width,  
                set_clear
```

# Cross-Reference Routines

## 8.3 \$CRFFIELD Macro

### label

The address of the field descriptor table generated as a result of this set of \$CRFFIELD macro instructions. The label field can be omitted after the first macro of the set. These addresses correspond to the field descriptor table addresses in the \$CRFCTLTABLE macro.

### bit\_mask

A 16-bit mask. When the user enters a key or reference, the cross-reference routine stores flag information with the entry. When preparing the output line, LIB\$CRE\_OUTPUT performs an AND operation on the 16-bit mask in the field descriptor table with the flag stored with the entry. Any number of bit masks can be defined for a field. \$CRFFIELD macro instructions are used to define multiple bit patterns for a flag field. The high-order bit is reserved to the cross-reference routines.

### fao\_string

The \$FAO command string. LIB\$CRE\_OUTPUT uses this string to determine the \$FAO format when formatting this field for output.

### field\_width

The maximum width of the output field.

### set\_clear

The indicator used to determine whether the bit mask is to be tested as set or clear when determining which flag to use. SET indicates test for set; CLEAR indicates test for clear.

You can use the argument names shown here as keywords in your program.

In the following example, one bit pattern is defined twice; once indicating a string that is to be printed if the pattern is set, and once indicating that spaces are to appear if the pattern is clear.

```
$CRFFIELD BIT_MASK=SYM$M_REL, FAO_STRING=3_\\##_\\,-  
SET_CLEAR=CLEAR, FIELD_WIDTH=2  
$CRFFIELD BIT_MASK=SYM$M_REL, FAO_STRING=_\\-R_\\,-  
SET_CLEAR=SET, FIELD_WIDTH=2
```

If more than one set of flags is defined for a field, each FAO string must print the same number of characters; otherwise, the output is not aligned in columns.

The fields for the symbol name, symbol value, and references are always formatted using the first descriptor in the corresponding table.

---

## 8.4 \$CRFFIELDEND Macro

The \$CRFFIELDEND macro instruction marks the end of a set of macros that describe one field of the output line. It is used once to end each set of field descriptors. It has the following format:

```
$CRFFIELDEND
```

# Cross-Reference Routines

## 8.5 Cross-Reference Output

### 8.5 Cross-Reference Output

LIB\$CRF\_OUTPUT can format output lines for three types of cross-reference listings:

- 1 A summary of symbol names and their values, as illustrated in Figure 8-2.
- 2 A summary of symbol names, their values, and the names of modules that refer to the symbol, as illustrated in Figure 8-3.
- 3 A summary of symbol names, their values, the name of the defining module, and the names of those modules that refer to the symbol, as illustrated in Figure 8-4.

**Figure 8-2 Summary of Symbol Names and Values**

```
+-----+
! Symbols By Name !
+-----+
```

Symbol	Value	Symbol	Value
BAS\$INSTR	000020B0-RU	BAS\$SCRATCH	00002308-RU
BAS\$IN_D_R	000021F0-RU	BAS\$STATUS	00002338-RU
BAS\$IN_F_R	000021E8-RU	BAS\$STR_D	000020C0-RU
BAS\$IN_L_R	000021E0-RU	BAS\$STR_F	000020B8-RU
BAS\$IN_T_DX	000021F8-RU	BAS\$STR_L	000020C8-RU
BAS\$IN_W_R	000021D8-RU	BAS\$UNLOCK	00002310-RU
BAS\$IO_END	000021D0-RU	BAS\$UPDATE	000022E8-RU
BAS\$LINKAGE	00001674-R	BAS\$UPDATE_COUN	000022F0-RU
BAS\$LINPUT	000021A8-RU	BAS\$VAL_D	00002110-RU
BAS\$MAT_INPUT	00002268-RU	BAS\$VAL_F	00002108-RU

ZK-1973-84

**Figure 8-3 Summary of Symbol Names, Values, and Name of Referring Modules**

Symbol	Value	Referenced By ...
BAS\$K_DIVBY_ZER	0000003D	ALLGBL      BAS\$error BAS\$POWDJ    BAS\$POWII BAS\$POWRJ    BAS\$POWRR
BAS\$K_DUPKEYDET	00000086	ALLGBL      BAS\$\$SIGNAL_IO
BAS\$K_ENDFILDEV	0000000B	ALLGBL      BAS\$\$REC_PROC
BAS\$K_ENDOF_STA	0000006C	BAS\$\$UDF_RL ALLGBL

ZK-1974-84

# Cross-Reference Routines

## 8.5 Cross-Reference Output

**Figure 8-4 Summary Indicating Defining Module**

Symbol	Value	Defined By	Referenced By ...
LIB\$FREE_VM	0001E185-R	LIB\$VM	ALLGBL BAS\$MARGIN BAS\$XLATE FOR\$VM STR\$APPEND STR\$DUPL_CHAR STR\$REPLACE
LIB\$GET_COMMAND	0001E2B0-R	LIB\$GET_INPUT	ALLGBL
LIB\$GET_COMMON	0001E4D6-R	LIB\$COMMON	ALLGBL

ZK-1971-84

Regardless of the format of the output, LIB\$CRE\_OUTPUT considers the output line to consist of the following six different field types:

- 1 KEY1 is the first field in the line. It contains a symbol name.
- 2 KEY2 is the second field in the line. It contains a set of flags (for example,-R) providing information about the symbol.
- 3 VAL1 is the third field in the line. It contains the value of the symbol.
- 4 VAL2 is the fourth field in the line. It contains a set of flags describing VAL1.
- 5 REF1 and REF2 fields. Within each REF1 and REF2 pair, REF1 provides a set of flags and REF2 provides the name of a module that references the symbol.

Any of these fields can be omitted from the output as shown in the following figure.

Symbol	Value	Symbol	Value
-----	-----	-----	-----
BAS\$INSTR	000020B0-RU	BAS\$SCRATCH	00002308-RU
↑	↑           ↑	↑	↑           ↑
KEY1	VAL1    VAL2	KEY1	VAL1    VAL2

Symbol	Value	Defined By	Referenced By ...
-----	-----	-----	-----
LIB\$FREE_VM	0001E185-R	LIB\$VM	ALLGBL
↑	↑           ↑	↑	↑
KEY1	VAL1    VAL2	REF2 (CRF\$K_DEF)	REF2 (CRF\$K_REF)

ZK-1972-84

### 8.6 Example

The VAX Linker uses the cross-reference routines to generate cross-reference listings. This section uses the linker's code as an example of using the cross-reference routines in a MACRO program.

#### 8.6.1 Defining Control Tables

Cross-reference routines use two control tables:

- The symbol-by-name table
- The symbol-by-value table

First, the linker uses the \$CRFCTLTABLE macro to set up the characteristics and fields of the symbol-by-name table. This table will list symbols by name and provide a cross-reference synopsis. The table is set up as follows:

LNK\$NAMTAB:

```
$CRFCTLTABLE KEYTYPE=ASCIC,ERROR=LNK$ERR_RTN,
OUTPUT=LNK$MAPOUT,KEY1TABLE=LNK$KEY1,
KEY2TABLE=LNK$KEY2,VAL1TABLE=LNK$VAL1,
VAL2TABLE=LNK$VAL2,REF1TABLE=LNK$REF1,
REF2TABLE=LNK$REF2
```

LNK\$NAMTAB	Names the address of the control table
KEYTYPE=ASCIC	Specifies that the keys are counted ASCII strings (that is, symbol names)
ERROR=LNK\$ERR_RTN	Indicates that LNK\$ERR_RTN is the address of the routine to be executed in case of error
OUTPUT=LNK\$MAPOUT	Names LNK\$MAPOUT as the address of the user-supplied routine that prints the formatted table

The remaining arguments provide the addresses of the field descriptor tables.

After setting up the control tables, the linker defines each field of the cross-reference output line, using the \$CRFFIELD macro. After each set of definitions for a field, it calls \$CRFFIELDEND to mark the end of the field.

Note particularly the following two features of this set of definitions.

- The definition of LNK\$VAL2 describes a flag to be associated with VAL1. The definition contains alternative bit patterns, depending on the bit mask. When an entry is made to the table, the entry contains flag information. Then, when LIB\$CRF\_OUTPUT is called to format the data, the routine checks each entry, matching the flags argument against the bit masks specified in the control table. When LIB\$CRF\_OUTPUT finds a match, it uses that definition to determine the format of the entry in the output table. For example, BIT\_MASK=SYM\$M\_DEF marks an entry as the defining reference. The corresponding VAL1 entry is placed in the output table with an asterisk in its flags field.
- The FAO control strings are defined to produce an output of the maximum character size for each field. This ensures that the columns will line up correctly in the output. For example, !15AC produces the variable symbol name left-aligned and right-filled with spaces. Another example is the three sets of characters to be printed for field VAL2. Each

# Cross-Reference Routines

## 8.6 Example

FAO control string produces two characters, which is the maximum size of the field.

```
LNK$KEY1:
    $CRFFIELD      BIT_MASK=0, FAO_STRING=!15AC\,-
                  SET_CLEAR=SET, FIELD_WIDTH=15
    $CRFFIELDEND
LNK$KEY2:
    $CRFFIELD      BIT_MASK=0, FAO_STRING=\ \,-
                  SET_CLEAR=SET, FIELD_WIDTH=1
    $CRFFIELDEND
LNK$VAL1:
    $CRFFIELD      BIT_MASK=0, FAO_STRING=!XL\,-
                  SET_CLEAR=SET, FIELD_WIDTH=8
    $CRFFIELDEND
LNK$VAL2:
    $CRFFIELD      BIT_MASK=0, FAO_STRING=!2* \,-
                  SET_CLEAR=SET, FIELD_WIDTH=2
    $CRFFIELD      BIT_MASK=SYM$M_REL, FAO_STRING=-R\,-
                  SET_CLEAR=SET, FIELD_WIDTH=2
    $CRFFIELD      BIT_MASK=SYM$M_DEF, FAO_STRING=-*\,-
                  SET_CLEAR=CLEAR, FIELD_WIDTH=2
    $CRFFIELDEND
LNK$REF1:
    $CRFFIELD      BIT_MASK=0, FAO_STRING=!6* \,-
                  SET_CLEAR=SET, FIELD_WIDTH=6
    $CRFFIELD      BIT_MASK=SYM$M_WEAK, FAO_STRING=!3* WK-\,-
                  SET_CLEAR=SET, FIELD_WIDTH=6
    $CRFFIELDEND
LNK$REF2:
    $CRFFIELD      BIT_MASK=0, FAO_STRING=!16AC\,-
                  SET_CLEAR=SET, FIELD_WIDTH=16
    $CRFFIELDEND
```

After initializing the symbol-by-name table, the linker sets up a second control table. This table defines the output for a symbol-by-value synopsis. For this output, the value fields are eliminated. The symbols having this value are entered as reference indicators. None is specified as the defining reference. The control table uses the field descriptors set up previously. The following macro instructions are used:

```
LNK$VALTAB:
    $CRFCTLTABLE  KEYTYPE=BIN_U32, ERROR=LNK$ERR_RTN,-
                  OUTPUT=LNK$MAPOUT, KEY1TABLE=LNK$VAL1,-
                  KEY2TABLE=LNK$VAL2, VAL1TABLE=0,-
                  VAL2TABLE=0, REF1TABLE=LNK$REF1,-
                  REF2TABLE=LNK$REF2
```

### 8.6.2 Inserting Table Information

After initializing the format data for the symbol tables, the linker enters data into the cross-reference tables by calling LIB\$CRF\_INS\_KEY.

# Cross-Reference Routines

## 8.6 Example

As the linker processes the first object module, MAPINITIAL, it encounters a symbol definition for \$MAPFLG. The following is an example of a call to enter the symbol MAPINITIAL as a key in the cross-reference symbol table:

```
PUSHAB VALUE_FLAGS
PUSHAB VALUE_ADDR
PUSHAB SYMBOL_ADDR
PUSHAB LNK$NAMTAB
CALLS #4,G^LIB$CRF_INS_KEY
```

LNK\$NAMTAB Is the address of the control table  
SYMBOL\_ADDR Is the address of the counted ASCII string \$MAPFLG  
VALUE\_ADDR Is the address of the symbol value  
VALUE\_FLAGS Is the address of a word whose bits are used to select special characters to print beside the value

The linker then calls LIB\$CRF\_INS\_REF to process the defining reference indicator:

```
DEF: .LONG CRF$K_DEF
      PUSHAB DEF
      PUSHAB REF_FLAGS
      PUSHAB REF_ADDR
      PUSHAB SYMBOL_ADDR
      PUSHAB LNK$NAMTAB
      CALLS #5,G^LIB$CRF_INS_REF
```

LNK\$NAMTAB Is the address of the control table  
SYMBOL\_ADDR Is the address of the counted string \$MAPFLG  
REF\_ADDR Is the address of the referrer's counted ASCII string  
REF\_FLAGS Is the address of a word whose bits are used to select special characters to print beside the reference

Further on in the input module, the linker encounters a global symbol reference to CS\$GBL. The call to store data for this reference is as follows:

```
REF: .LONG CRF$K_REF
      PUSHAB REF
      PUSHAB REF_FLAGS
      PUSHAB REF_ADDR
      PUSHAB SYMBOL_ADDR
      PUSHAB LNK$NAMTAB
      CALLS #5,G^LIB$CRF_INS_REF
```

The arguments are similar to the previous example, except for CRF\$K\_REF, which indicates that this is not the defining reference.

After it has performed symbol relocation for the module being linked, the linker calls LIB\$CRF\_INS\_REF to build a table ordered by value.



# Cross-Reference Routines

## 8.6 Example

PUSHAB	REF
PUSHAB	REF_FLAGS
PUSHAB	REF_ADDR
PUSHAB	VAL_ADDR
PUSHAB	LNK\$VALTAB
CALLS	#5,G^LIB\$CRF_INS_REF

LNK\$VALTAB Is the address of the control table for the symbol synopsis by value

VAL\_ADDR Is the address of the value (binary longword key)

REF\_ADDR Is the address of the symbol name having the value contained in VAL\_ADDR

REF\_FLAGS Is the address of a word whose bits are used to select special characters to print beside the value

CRF\$K\_REF Is the indicator that this is not a defining reference

### 8.6.3 Formatting Information for Output

After all input modules are processed, the linker requests the information for the map. It calls LIB\$CRF\_OUTPUT once for each type of output. The following MACRO example illustrates a call to list the symbols and their values. Three calls are illustrated here.

```
LNWID: .LONG 132
LNSP1: .LONG LINES_PAGE1
LNSOP: .LONG LINES_OTHR_PAGE
SAVE: .LONG CRF$K_SAVE
VAL: .LONG CRF$K_VALUES
PUSHAB VAL
PUSHAB SAVE
PUSHAB LNSOP
PUSHAB LNSP1
PUSHAB LNWID
PUSHAB LNK$NAMTAB
CALLS #6,G^LIB$CRF_OUTPUT
```

In this example, CRF\$K\_VALUES means that no reference indicators are to be printed, while CRF\$K\_SAVE means that the cross-reference table is to be saved. It is also possible to list all cross-reference data. The type of output produced by this call is shown in Section 8.5, Figure 8-2.

The following call produces such a summary and releases the storage at the same time:

```
LNWID: .LONG 132
LNSP1: .LONG LINES_PAGE1
LNSOP: .LONG LINES_OTHR_PAGE
DELETE: .LONG CRF$K_DELETE
DEFREF: .LONG CRF$K_DEF_REF
PUSHAB DELETE
PUSHAB DEFREF
PUSHAB LNSOP
PUSHAB LNSP1
PUSHAB LNWID
PUSHAB LNK$NAMTAB
CALLS #6,G^LIB$CRF_OUTPUT
```

The type of output produced by this call is shown in Section 8.5, Figure 8-4.

# Cross-Reference Routines

## 8.6 Example

CRF\$K\_DEFS\_REFS indicates that the first two reference fields are used for the defining references, and CRF\$K\_DELETE indicates that the table is deleted.

Another call is made to list the symbol by value synopsis, as follows:

```
LNWID:      .LONG      132
LNSP1:      .LONG      LINES_PAGE1
LNSOP:      .LONG      LINES_OTHR_PAGE
VALREF:     .LONG      CRF$K_VALS_REF
DELETE:     .LONG      CRF$K_DELETE
            PUSHAB    DELETE
            PUSHAB    VALREF
            PUSHAB    LNSOP
            PUSHAB    LNSP1
            PUSHAB    LNWID
            PUSHAB    LNK$VALTAB
            CALLS     #6,G^LIB$CRF_OUTPUT
```

This is similar to the previous call in that it produces a complete cross-reference output by value, but it does not have the defining reference fields.

---

## 8.7 How to Link to the Cross-Reference Shareable Image

The cross-reference routines are located in a shareable image CRFSHR.EXE. This shareable image is part of the default system shareable image library, SYS\$LIBRARY:IMAGELIB.OLB. For this reason, the cross-reference routines are automatically included in your image, unless you specify /NOSYSHR in the LINK command. If you have specified /NOSYSHR and you want to include CRFSHR.EXE, your LINK command must include the following:

```
SYS$LIBRARY:IMAGELIB/INCLUDE=CRFSHR
```



---

## **LIB\$ Reference Section**

This section provides detailed discussions of the routines provided by the VMS RTL Library (LIB\$) Facility.



---

## LIB\$ADAWI Add Adjacent Word with Interlock

The Add Adjacent Word with Interlock routine allows the user to perform an interlocked add operation using an aligned word.

---

**FORMAT**            **LIB\$ADAWI** *add ,sum ,result*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:           **longword (unsigned)**  
                           access:       **write only**  
                           mechanism:   **by value**

---

**ARGUMENTS**        ***add***  
                           VMS usage: **word\_signed**  
                           type:       **word (signed)**  
                           access:     **read only**  
                           mechanism: **by reference**

The addend operand to be added to the value of **sum**. The **add** argument is the address of a signed word that contains the addend operand.

***sum***  
 VMS usage: **word\_signed**  
 type:       **word integer (signed)**  
 access:     **modify**  
 mechanism: **by reference**

The word to which **add** is added. The **sum** argument is the address of a signed word integer containing this value.

***result***  
 VMS usage: **word\_signed**  
 type:       **word integer (signed)**  
 access:     **modify**  
 mechanism: **by reference**

The result of adding **add** and **sum**. The **result** argument is the address of a signed word integer containing the result.

---

**DESCRIPTION**        LIB\$ADAWI allows the user to perform an interlocked add operation using an aligned word, and makes the VAX ADAWI instruction available as a callable routine. This routine also enables the user to implement synchronization primitives for multiprocessing.

The add operation is interlocked against similar operations on other processors in a multiprocessor environment. The destination must be aligned on a word boundary; that is, bit 0 of the address of the sum operand must be 0.

# LIB\$ADAWI

If the addend and the sum operand overlap, the result of the addition, the value of the result parameter, and the associated condition codes are unpredictable.

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Routine successfully completed.

LIB\$\_INTOVF

Integer overflow error.

---

## LIB\$ADD\_TIMES Add Two Quadword Times

The Add Two Quadword Times routine adds two VMS internal time format times.

---

**FORMAT**            **LIB\$ADD\_TIMES**    *time1 ,time2 ,resultant-time*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:           **longword (unsigned)**  
                           access:       **write only**  
                           mechanism:   **by value**

---

**ARGUMENTS**        ***time1***  
                           VMS usage: **date\_time**  
                           type:       **quadword (unsigned)**  
                           access:     **read only**  
                           mechanism: **by reference**

First time that LIB\$ADD\_TIMES adds to the second time. The **time1** argument is the address of an unsigned quadword containing the first time to be added. **Time1** may be either a delta time or an absolute time; however, at least one of the arguments, **time1** or **time2**, must be a delta time.

***time2***  
                           VMS usage: **date\_time**  
                           type:       **quadword (unsigned)**  
                           access:     **read only**  
                           mechanism: **by reference**

Second time that LIB\$ADD\_TIMES adds to the first time. The **time2** argument is the address of an unsigned quadword containing the second time to be added. **Time2** may be either a delta time or an absolute time; however, at least one of the arguments, **time1** or **time2**, must be a delta time.

***resultant-time***  
                           VMS usage: **date\_time**  
                           type:       **quadword (unsigned)**  
                           access:     **write only**  
                           mechanism: **by reference**

The result of adding **time1** and **time2**. The **resultant-time** argument is the address of an unsigned quadword containing the result. If both **time1** and **time2** are delta times, then **resultant-time** is a delta time. Otherwise, **resultant-time** is an absolute time.

---

**DESCRIPTION**        LIB\$ADD\_TIMES adds two VMS internal times. It can add two delta times or a delta time and an absolute time. LIB\$ADD\_TIMES cannot add two absolute times. Delta times must be less than 10,000 days.



# LIB\$ADD\_TIMES

---

**CONDITION  
VALUES  
RETURNED**

LIB\$\_NORMAL

Routine successfully completed.

LIB\$\_IVTIME

Invalid time.

LIB\$\_ONEDELTIM

At least one delta time is required.

LIB\$\_WRONUMARG

Incorrect number of arguments.



# LIB\$ADDX

## *array-length*

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Length in longwords of the arrays to be operated on; each array is of length **len**. The **len** argument is the address of a signed longword integer containing the length. **Len** must not be negative. This is an optional argument. If omitted, the default is 2.

---

## DESCRIPTION

LIB\$ADDX adds two signed two's complement integers of arbitrary length. The integers are located in arrays of longwords. The higher addresses of these longwords contain the higher-precision parts of the values. The highest-addressed longword contains the sign and 31 bits of precision. The remaining longwords contain 32 bits of precision in each. The number of longwords in each array is specified in the optional argument, **len**. The default length is two, which corresponds to the VAX quadword data type.

Any two or all three of the first three arguments can be the same.

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Routine successfully completed.

SS\$\_INTOVF

Integer overflow. The result is correct, except that the sign bit is lost.

---

## EXAMPLE

```
C+
C This FORTRAN example program illustrates the use
C of LIB$ADDX.
C-

      INTEGER A(2),B(2),C(2),RETURN
      DATA A/'00000001'x,'7FFF407F'x/
      DATA B/'FFFFFFFF'x,'8000BF80'x/

C+
C The highest addressed longword of "A" is A(2).
C So, "A" represents the integer value ('7FFF407F'x) * 16**7 + 1.
C That is, A(2) is 576447592255193089.
C "B" is the twos complement representation of "-A".
C-

      RETURN = LIB$ADDX(A,B,C)
      TYPE *, 'Let A = 576447592255193089.'
      TYPE *, 'Then A + B is 0.'
      TYPE 1,C(2),C(1)
1     FORMAT(' "A" - "A" is ',1H',I1,I1,3H'x.)
      TYPE *, 'Note that C is C(2) concatenated with C(1).'
```

```
C+
C Let "A" have the value 72057594037927937 = '1000000000000001'x.
C Let "B" have the value 4294967295       = '00000000FFFFFFFF'x.
C-
```

```

      A(1) = '00000001'x
      A(2) = '10000000'x
      B(1) = 'FFFFFFFF'x
      B(2) = '00000000'x
C+
C Then "A" + "B" is 72057598332895232.
C-
      RETURN = LIB$ADDX(A,B,C)
      TYPE *, ' '
      TYPE *, 'LET A = 72057594037927937 and B = 4294967295'
      TYPE *, 'Then A + B is ',C
      TYPE 2,C(2),C(1)
2  FORMAT(' 72057598332895232 is represented as ',1H',Z8,Z8,3H'x.)
      TYPE *, 'Recall that 72057598332895232 is C(2) concatenated
1 with C(1).'
```

```

      END
```

This FORTRAN example demonstrates how to call LIB\$ADDX. The output generated by this program is as follows:

```

Let A = 576447592255193089.
Then A + B is 0.
"A" - "A" is '00'x.
Note that C is C(2) concatenated with C(1).
LET A = 72057594037927937 and B = 4294967295
Then A + B is          0  268435457
72057598332895232 is represented as '10000001      0'x.
Recall that 72057598332895232 is C(2) concatenated with C(1).
```



**DESCRIPTION**

LIB\$ANALYZE\_SDESC extracts the length and the address at which the data starts for a variety of string descriptor classes. Following is a description of the classes of string descriptors.

Class	Description	Restrictions/Notes
A	Array	DSC\$_ARSIZE must be less than 65536 bytes.
D	Decimal string	Treated as Class S.
NCA	Noncontiguous array	Same as Class A.
S	Scalar, string	None.
SD	Decimal scalar	Treated as Class S.
VS	Varying string	Length returned is CURLEN.
Z	Unspecified	Treated as Class S.

See STR\$ANALYZE\_SDESC for a similar routine that signals an error rather than returning a status.

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_INVSTRDES	Invalid string descriptor. An array descriptor has an ARSIZE greater than 65,535 bytes, or the class is unsupported.

# LIB\$ASN\_WTH\_MBX

---

## LIB\$ASN\_WTH\_MBX Assign Channel with Mailbox

The Assign Channel with Mailbox routine assigns a channel to a specified device and associates a mailbox with the device. It returns both the device channel and the mailbox channel.

---

**FORMAT**            **LIB\$ASN\_WTH\_MBX**    *device-name*  
  *,maximum-message-size*  
  *,buffer-quota ,device-channel*  
  *,mailbox-channel*

---

**RETURNS**            VMS usage: **cond\_value**  
  type:            **longword (unsigned)**  
  access:         **write only**  
  mechanism:     **by value**

---

**ARGUMENTS**         ***device-name***  
  VMS usage: **device\_name**  
  type:            **character string**  
  access:         **read only**  
  mechanism:     **by descriptor**

Device name which LIB\$ASN\_WTH\_MBX passes to the \$ASSIGN service. The **device-name** argument is the address of a descriptor pointing to the device name.

***maximum-message-size***  
VMS usage: **longword\_signed**  
type:         **longword integer (signed)**  
access:       **read only**  
mechanism:   **by reference**

Maximum message size that can be sent to the mailbox; LIB\$ASN\_WTH\_MBX passes this argument to the \$CREMBX service. The **maximum-message-size** argument is the address of a signed longword integer containing this maximum message size.

***buffer-quota***  
VMS usage: **longword\_signed**  
type:         **longword integer (signed)**  
access:       **read only**  
mechanism:   **by reference**

Number of system dynamic memory bytes that can be used to buffer messages sent to the mailbox; LIB\$ASN\_WTH\_MBX passes this argument to the \$CREMBX service. The **buffer-quota** argument is the address of a signed longword integer containing this buffer quota.

***device-channel***

VMS usage: **word\_unsigned**  
 type: **word integer (unsigned)**  
 access: **write only**  
 mechanism: **by reference**

Device channel which LIB\$ASN\_WTH\_MBX receives from the \$ASSIGN service. The **device-channel** argument is the address of an unsigned word integer into which \$ASSIGN writes the device channel.

***mailbox-channel***

VMS usage: **channel**  
 type: **word integer (unsigned)**  
 access: **write only**  
 mechanism: **by reference**

Mailbox channel which LIB\$ASN\_WTH\_MBX receives from the \$CREMBX service. The **mailbox-channel** argument is the address of an unsigned word integer into which \$CREMBX writes the mailbox channel.

---

**DESCRIPTION**

A mailbox is a virtual device used for communication between processes. A channel is the communication path that a process uses to perform I/O operations to a particular device. LIB\$ASN\_WTH\_MBX assigns a channel to a device and associates a mailbox with the device. It returns both the device channel and the mailbox channel to the mailbox.

Normally, a process calls the \$CREMBX system service to create a mailbox and assign a channel and logical name to it. Any process running in the same job and using the same logical name uses the same mailbox.

LIB\$ASN\_WTH\_MBX associates the physical mailbox name with the channel assigned to the device. To create a temporary mailbox for itself and other processes cooperating with it, your program calls LIB\$ASN\_WTH\_MBX. The Run-Time Library routine assigns the channel and creates the temporary mailbox by using the system services \$GETDVIW, \$ASSIGN, and \$CREMBX. Instead of a logical name, the mailbox is identified by a physical device name of the form MBcu. The physical device name MBcu is made up of the following elements:

MB Indicates that the device is a mailbox  
 c Is the controller  
 u Is the unit number

The routine returns this device name to the calling program, which then must pass the mailbox channel to the other program(s) with which it cooperates. In this way, the cooperating processes access the mailbox by its physical name, instead of by a logical name.

The calling program passes the routine a device name, which specifies the device to which the channel is to be assigned. For this argument (called **device-name**), you may use a logical name. If you do so, the routine attempts one level of logical name translation.

The privilege restrictions and process quotas required for using this routine are those required by the \$GETDVIW, \$CREMBX, and \$ASSIGN system services.



# LIB\$ASN\_WTH\_MBX

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Routine successfully completed.

Any condition value returned by the called system services \$ASSIGN, \$CREMBX, \$GETDVI, or the RTL routines LIB\$GET\_EF and LIB\$FREE\_EF.

---

## LIB\$AST\_IN\_PROG AST in Progress

The AST in Progress routine indicates whether an AST is currently in progress.

---

**FORMAT**                    **LIB\$AST\_IN\_PROG**

---

**RETURNS**

VMS usage: **boolean**  
 type:            **boolean**  
 access:         **write only**  
 mechanism:     **by value**

Truth value that indicates whether an AST is currently in progress (value=1) or not (value=0).

---

**ARGUMENTS**            *None.*

---

**DESCRIPTION**

An asynchronous system trap (AST) is a VMS mechanism for providing a software interrupt when an external event occurs, such as the user typing CTRL/C. When an external event occurs, VMS interrupts the execution of the current process and calls a routine that you supply. While that routine is active, the AST is said to be in progress, and the process is said to be executing at AST level. When your AST routine returns control to the original process, the AST is no longer active and execution continues where it left off.

LIB\$AST\_IN\_PROG indicates to the calling program whether an AST is currently in progress. Your program can call LIB\$AST\_IN\_PROG to determine whether it is executing at AST level, and then take appropriate action. This routine is useful if you are writing AST-reentrant code, which takes different actions depending on whether an AST is in progress. For example, the routine might have two separate statically allocated storage areas, one for AST level and one for non-AST level.

---

**CONDITION  
VALUES  
RETURNED**

Any condition values returned by LIB\$FREE\_EF, LIB\$GET\_EF, or SYS\$GETJPI.

# LIB\$AST\_IN\_PROG

---

## EXAMPLE

```
PROGRAM AST_IN_PROGRESS(INPUT, OUTPUT);
FUNCTION LIB$AST_IN_PROG : INTEGER; EXTERN;
VAR
  ASTVALUE : INTEGER;
BEGIN
  ASTVALUE := LIB$AST_IN_PROG;
  CASE ASTVALUE OF
    0 : Writeln('AN AST IS NOT IN PROGRESS');
    1 : Writeln('AN AST IS IN PROGRESS');
  END { of the case statement }
END.
```

This Pascal program determines whether or not an AST is in progress.

---

## LIB\$ATTACH Attach Terminal to Process

The Attach Terminal to Process routine requests the calling process's Command Language Interpreter (CLI) to detach the terminal of the calling process and to reattach it to a different process.

---

**FORMAT**            **LIB\$ATTACH** *process-id*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:       **longword (unsigned)**  
                           access:     **write only**  
                           mechanism: **by value**

---

**ARGUMENT**        ***process-id***  
                           VMS usage: **process\_id**  
                           type:       **longword integer (unsigned)**  
                           access:     **read only**  
                           mechanism: **by reference**

Identification of the process to which LIB\$ATTACH requests the calling process to attach its terminal. The **process-id** argument is the address of an unsigned longword integer containing the process identification. The specified process must be currently detached (by means of a SPAWN or ATTACH command, or by a call to LIB\$SPAWN or LIB\$ATTACH) and must be part of the caller's job.

---

**DESCRIPTION**     LIB\$ATTACH requests the calling process's Command Language Interpreter (CLI) to detach the terminal of the calling process and to reattach it to a different process. The calling process then hibernates. LIB\$ATTACH provides the same function as the DCL command ATTACH. For more information, see the *VMS DCL Dictionary*.

LIB\$ATTACH is supported for use with the DCL CLI. If used with the Monitor Control Routine (MCR) CLI, the error status LIB\$\_NOCLI will be returned. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In such cases the error status LIB\$\_NOCLI is returned.

# LIB\$ATTACH

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
SS\$_NONEXPR	Nonexistent process. The process specified by <b>process-id</b> does not exist.
LIB\$_ATTREQREF	Attach request refused. The specified process could not be attached to. Either it was not detached or did not belong to the caller's job.
LIB\$_NOCLI	No CLI present to perform function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL CLI, please report the problem to DIGITAL by means of a Software Performance Report (SPR).

---

## LIB\$BBCCI Test and Clear Bit with Interlock

The Test and Clear Bit with Interlock routine tests and clears a selected bit under memory interlock. LIB\$BBCCI makes the VAX BBCCI instruction available as a callable routine.

---

**FORMAT**            **LIB\$BBCCI**    *position ,bit-zero-address*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                           type:        **longword (unsigned)**  
                           access:     **write only**  
                           mechanism: **by value**

State of the bit before it was cleared by LIB\$BBCCI; 1 if the bit was previously set and 0 if the bit was previously clear.

---

**ARGUMENTS**        ***position***  
                           VMS usage: **longword\_signed**  
                           type:        **longword integer (signed)**  
                           access:     **read only**  
                           mechanism: **by reference**

Bit position, relative to **bit-zero-address**, of the bit which LIB\$BBCCI tests and clears. The **position** argument is the address of a signed longword integer containing the bit position. A position of zero denotes the low-order bit of the byte base. The bit position is equal to the offset of the bit chosen from the base position. This offset may span the entire range of a signed longword integer; negative offsets access bits in lower-addressed bytes.

***bit-zero-address***  
                           VMS usage: **unspecified**  
                           type:        **unspecified**  
                           access:     **modify**  
                           mechanism: **by reference**

Address of the byte containing bit zero of the field that LIB\$BBCCI references. The **bit-zero-address** argument is the location of the base position. The bit that LIB\$BBCCI tests and clears is **position** bits offset from the low bit of **bit-zero-address**.

---

**DESCRIPTION**        The single bit specified by **position** and **bit-zero-address** is tested, the previous state of the bit remembered, and the bit cleared. The reading of the state of the bit and its clearing are interlocked against similar operations by other processors or devices in the system. The remembered previous state of the bit is then returned as the function value of LIB\$BBCCI.

For more information, see the *VAX Architecture Reference Manual*.

# LIB\$BBCCI

---

**CONDITION**      *None.*  
**VALUES**  
**RETURNED**

---

## EXAMPLE

```
C+
C This FORTRAN program demonstrates the use of
C LIB$BBCCI.
C-

      INTEGER*4 STATES(4)          ! 128 shared state bits
      COMMON /STATES/ STATES      ! Could be shared memory
      LOGICAL*4 LIB$BBCCI

      IF (LIB$BBCCI (42, STATES)) THEN
         TYPE *, 'State bit 42 was set'
      ELSE
         TYPE *, 'State bit 42 was clear'
      END IF
      END
```

This FORTRAN example tests and clears bit 42 of array STATES, which is in a COMMON area (possibly shared between two processors).

The output generated by this program is as follows:

```
§ RUN STATE
State bit 42 was clear.
```

---

## LIB\$BBSSI Test and Set Bit with Interlock

The Test and Set Bit with Interlock routine tests and sets a selected bit under memory interlock. LIB\$BBSSI makes the VAX BBSSI instruction available as a callable routine.

---

**FORMAT**                    **LIB\$BBSSI** *position ,bit-zero-address*

---

**RETURNS**                    VMS usage: **longword\_unsigned**  
                                   type:            **longword (unsigned)**  
                                   access:        **write only**  
                                   mechanism:    **by value**

The state of the bit before it was set by LIB\$BBSSI; 1 if it was previously set, 0 if it was previously clear.

---

### ARGUMENTS

***position***  
 VMS usage: **longword\_signed**  
 type:        **longword integer (signed)**  
 access:     **read only**  
 mechanism: **by reference**

Bit position, relative to **bit-zero-address**, of the bit which LIB\$BBSSI tests and sets. The **position** argument is the address of a signed longword integer containing the bit position. A position of zero denotes the low-order bit of the byte base. The bit position is equal to the offset of the bit chosen from the base position. This offset may span the entire range of a signed longword integer; negative offsets access bits in lower-addressed bytes.

***bit-zero-address***  
 VMS usage: **unspecified**  
 type:        **unspecified**  
 access:     **modify**  
 mechanism: **by reference**

Address of the byte containing bit zero of the field that LIB\$BBSSI references. The **bit-zero-address** argument is the location of the base position. The bit that LIB\$BBSSI tests and sets is **position** bits offset from the low bit of **bit-zero-address**.

---

### DESCRIPTION

The single bit specified by **position** and **bit-zero-address** arguments is tested, the previous state of the bit remembered, and the bit set. The reading of the state of the bit and its setting are interlocked against similar operations by other processors or devices in the system. The remembered previous state of the bit is then returned as the function value of LIB\$BBSSI.

For more information, see the *VAX Architecture Reference Manual*.



# LIB\$BBSSI

---

**CONDITION**      *None.*  
**VALUES**  
**RETURNED**

---

## EXAMPLE

```
C+
C This FORTRAN example program demonstrates
C the use of LIB$BBSSI.
C-
```

```
INTEGER*4 STATES(4)      ! 128 shared state bits
COMMON /STATES/ STATES  ! Could be shared memory
LOGICAL*4 LIB$BBSSI

IF (LIB$BBSSI (104, STATES)) THEN
  TYPE *, 'State bit 104 was set'
ELSE
  TYPE *, 'State bit 104 was clear'
END IF
END
```

This FORTRAN example tests and sets bit 104 of array STATES, which is in a COMMON storage area (possibly shared between two processors).

The output generated by this program is as follows:

```
$ RUN STATEB
State bit 104 was clear.
```

---

## LIB\$CALLG Call Routine with General Argument List

The Call Routine with General Argument List routine calls a routine with an argument list specified as an array of longwords, the first of which is a count of the remaining longwords. LIB\$CALLG is a callable version of the VAX CALLG instruction.

---

**FORMAT**                    **LIB\$CALLG** *argument-list ,user-procedure*

---

**RETURNS**                VMS usage: **longword\_unsigned**  
                               type:            **longword (unsigned)**  
                               access:        **write only**  
                               mechanism:    **by value**

Return value, if any, of the called routine. This value is not changed by LIB\$CALLG.

---

**ARGUMENTS**            ***argument-list***  
                               VMS usage: **arg\_list**  
                               type:        **unspecified**  
                               access:     **read only**  
                               mechanism: **by reference, array reference**

Argument list which LIB\$CALLG uses to call the specified routine. The **argument-list** argument is the address of an array of longwords containing the argument list. The first longword must contain the count of the remaining longwords. The maximum value of the count is 255.

***user-procedure***  
 VMS usage: **procedure**  
 type:        **procedure entry mask**  
 access:     **function call (before return)**  
 mechanism: **by value**

routine which LIB\$CALLG calls with the specified argument list. The **user-procedure** argument is the address of the routine entry mask for this routine.

---

**DESCRIPTION**            LIB\$CALLG is useful for calling routines which accept variable-length argument lists when the number of arguments to be passed is not known until execution time. LIB\$CALLG can also be used to call such routines from strongly typed languages which require routines to be declared as having a fixed number of arguments.

For more information, see the *VAX Architecture Reference Manual*.

# LIB\$CALLG

---

**CONDITION**      *None.*  
**VALUES**  
**RETURNED**

---

## LIB\$CHAR Transform Byte to First Character of String

The Transform Byte to First Character of String routine transforms a single 8-bit ASCII character to an ASCII string consisting of a single character followed by trailing spaces, if needed, to fill out the string. The range of the input byte is 0 through 255.

---

**FORMAT**            **LIB\$CHAR** *one-character-string ,ascii-code*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:          **write only**  
                          mechanism:       **by value**

---

**ARGUMENTS**          ***one-character-string***  
                          VMS usage: **char\_string**  
                          type:            **character string**  
                          access:          **write only**  
                          mechanism:       **by descriptor**

ASCII character string consisting of a single character followed by trailing spaces, if needed, that LIB\$CHAR creates when it transforms the ASCII character code. The ***one-character-string*** argument is the address of a descriptor pointing to the character string that LIB\$CHAR writes.

### ***ascii-code***

VMS usage: **byte\_unsigned**  
type:            **byte (unsigned)**  
access:          **read only**  
mechanism:       **by reference**

Single 8-bit ASCII character code that LIB\$CHAR transforms to an ASCII string. The ***ascii-code*** argument is the address of an unsigned byte containing the ASCII character code.

---

**DESCRIPTION**        LIB\$CHAR is the inverse of LIB\$ICHAR. (See the description of LIB\$ICHAR.) LIB\$CHAR is not a binary-to-ASCII conversion routine. LIB\$CHAR merely interprets ***ascii-code*** as an ASCII character code and converts it to a string.

# LIB\$CHAR

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Routine successfully completed, but the string was truncated. The fixed-length destination string could not contain all of the characters.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.

# LIB\$CONVERT\_DATE\_STRING

---

## LIB\$CONVERT\_DATE\_STRING Convert Date String to Quadword

The Convert Date String to Quadword routine converts an absolute date string into a VMS internal format date-time quadword. That is, given an input date/time string of a specified format, LIB\$CONVERT\_DATE\_STRING converts this string to a VMS internal format time.

---

**FORMAT**            **LIB\$CONVERT\_DATE\_STRING**    *date-string ,date-time*  
    *[,user-context]*  
    *[,flags] [,defaults]*  
    *[,defaulted-fields]*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:        **write only**  
                          mechanism:    **by value**

---

**ARGUMENTS**        ***date-string***  
                         VMS usage: **time\_name**  
                         type:            **character-coded text string**  
                         access:        **read only**  
                         mechanism:    **by descriptor**

Date string that specifies the absolute time to be converted to an internal system time. The ***date-string*** argument is the address of a descriptor pointing to this date string. This string must have a format corresponding to the currently defined input format, or it must be one of the relative day strings YESTERDAY, TODAY, or TOMORROW, or their equivalents in the currently selected language.

***date-time***  
VMS usage: **date\_time**  
type:        **quadword (unsigned)**  
access:     **write only**  
mechanism: **by reference**

Receives the converted time. The ***date-time*** argument is the address of an unsigned quadword that contains this VMS internal format converted time.

***user-context***  
VMS usage: **context**  
type:        **longword (unsigned)**  
access:     **modify**  
mechanism: **by reference**

# LIB\$CONVERT\_DATE\_STRING

Context variable that retains the translation context over multiple calls to this routine. The **user-context** argument is the address of an unsigned longword that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.

The **user-context** parameter is optional. However, if a context cell is not passed, the routine LIB\$CONVERT\_DATE\_STRING may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, DIGITAL recommends that users specify a different context cell for each calling thread.

## **flags**

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Specifies which date or time fields of the **date-string** argument might be omitted so that default values are applied. The **flags** argument is the address of a longword bit mask that contains these flags. A set bit indicates that the field may be omitted. The bit definitions for the mask correspond to the fields in a \$NUMTIM "timbuf" structure as follows:

Field	Bit Number	Mask
Year	0	1
Month	1	2
Day of month	2	4
Hours	3	8
Minutes	4	16
Seconds	5	32
Fractional seconds	6	64

Bits 7 through 31 must be zero and are reserved for use by DIGITAL. If this parameter is omitted, a default value of 120 (78H) is used, indicating that the time fields may be defaulted, but the date fields may not.

## **defaults**

VMS usage: **vector\_word\_unsigned**  
type: **word (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Supplies the defaults to be used for omitted fields. The **defaults** argument is the address of an array of unsigned words containing these default values. This array corresponds to a 7-word \$NUMTIM "timbuf" structure. If the **defaults** argument is omitted, the following defaults are applied:

- For the date group, the default is the current date.
- For the time group, the default is 00:00:00.00.

# LIB\$CONVERT\_DATE\_STRING

## *defaulted-fields*

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by reference**

Indicates which date or time fields have been defaulted. The **defaulted-fields** argument is the address of a longword bit mask that specifies these fields. The bit definitions are identical to those of the **flags** bit mask. A set bit indicates that the field was defaulted. Bits 7 through 31, which are reserved for use by DIGITAL, are zeroed.

---

## DESCRIPTION

LIB\$CONVERT\_DATE\_STRING converts an absolute date string into a VMS internal format date-time quadword. The input date string can either correspond to the format specified, or it can be the language equivalent of one of the relative date strings YESTERDAY, TODAY, or TOMORROW. The language to be used and the format in which to interpret the information are programmable using either of the following methods.

- The language and format are programmable at compile time through the use of the routine LIB\$INIT\_DATE\_TIME\_CONTEXT.
- The language and format can be determined at run time through the translation of the logical names SYS\$LANGUAGE and LIB\$DT\_INPUT\_FORMAT.

In general, if an application is reading text from internal storage, the language and input format should be specified at compile time. If this is the case, use the routine LIB\$INIT\_DATE\_TIME\_CONTEXT to specify the language and input format of your choice.

If an application is accepting text from a user, the logical name method of specifying language and format should be used. In this method, the user assigns equivalence names to the logical names SYS\$LANGUAGE and LIB\$DT\_INPUT\_FORMAT, thereby selecting the language and input format of the date and time at run time.

The calling program can choose to apply defaults for omitted fields in the date string. To do this, the **flags** argument is used to indicate which fields are to be defaulted, and the **defaults** argument is used to supply the default values. If the **defaults** argument is not supplied, the following default values are applied:

- For the date group, the default is the current date.
- For the time group, the default is 00:00:00.00.

Optionally, you can use the **defaulted-fields** argument to receive information on which input fields were omitted and thus accepted default values.



# LIB\$CONVERT\_DATE\_STRING

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_DEFFORUSE	Default format used; unable to determine desired format.
LIB\$_ENGLUSED	English used by default; unable to translate SYS\$LANGUAGE.
LIB\$_AMBDATTIM	Ambiguous date/time.
LIB\$_INCDATTIM	Incomplete date/time; missing fields with no defaults.
LIB\$_ILLFORMAT	Illegal format string; too many or not enough fields.
LIB\$_INVARG	Invalid argument; a required argument was not specified.
LIB\$_INVSTRDES	Invalid input string descriptor.
LIB\$_IVTIME	Invalid date/time.
LIB\$_REENTRANCY	Reentrancy detected.
LIB\$_UNRFORCOD	Unrecognized format code.
LIB\$_WRONUMARG	Wrong number of arguments.

Any condition value returned by LIB\$GET\_VM, LIB\$FREE\_VM, LIB\$FREE1\_DD, LIB\$SCOPY\_R\_DX, SYS\$NUMTIM, and SYS\$GETTIM.

---

## LIB\$CRC Calculate a Cyclic Redundancy Check (CRC)

The Calculate a Cyclic Redundancy Check routine calculates the cyclic redundancy check (CRC) for a data stream. LIB\$CRC makes the VAX CRC instruction available as a callable routine.

---

**FORMAT**            **LIB\$CRC** *crc-table ,initial-crc ,stream*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                           type:        **longword (unsigned)**  
                           access:     **write only**  
                           mechanism: **by value**

The computed cyclic redundancy check.

---

**ARGUMENTS**        ***crc-table***  
                           VMS usage: **vector\_longword\_signed**  
                           type:        **longword integer (signed)**  
                           access:     **read only**  
                           mechanism: **by reference, array reference**

The 16-longword cyclic redundancy check table, created by a call to LIB\$CRC\_TABLE. The **crc-table** argument is the address of a signed longword integer containing this table. Because this table is created by LIB\$CRC\_TABLE and then used as input in LIB\$CRC, your program must call LIB\$CRC\_TABLE before it calls LIB\$CRC.

***initial-crc***  
 VMS usage: **longword\_signed**  
 type:        **longword integer (signed)**  
 access:     **read only**  
 mechanism: **by reference**

Initial cyclic redundancy check. The **initial-crc** argument is the address of a signed longword integer containing the initial cyclic redundancy check.

***stream***  
 VMS usage: **char\_string**  
 type:        **character string**  
 access:     **read only**  
 mechanism: **by descriptor**

Data stream for which LIB\$CRC is calculating the CRC. The **stream** argument is the address of a descriptor pointing to the data stream.

# LIB\$CRC

---

## DESCRIPTION

Before your program can call LIB\$CRC, it must call LIB\$CRC\_TABLE. LIB\$CRC\_TABLE takes a polynomial as its input and builds the table that LIB\$CRC uses to calculate the CRC.

LIB\$CRC allows your high-level language program to use the CRC instruction, which calculates the Cyclic Redundancy Check. This instruction checks the integrity of a data stream by comparing its state at the sending point and the receiving point. Each character in the data stream is used to generate a value based on a polynomial. The values for each character are then added together. This operation is performed at both ends of the data transmission, and the two result values compared. If the results disagree, then an error occurred during the transmission.

See the *VAX Architecture Reference Manual* for a description of the algorithms used in computing the CRC.

---

## CONDITION VALUES RETURNED

*None.*

---

## EXAMPLE

For an example of using LIB\$CRC, refer to the BASIC example at the end of the description of LIB\$CRC\_TABLE.

---

## LIB\$CRC\_TABLE Construct a Cyclic Redundancy Check (CRC) Table

The Construct a Cyclic Redundancy Check Table routine constructs a 16-longword table that uses a cyclic redundancy check polynomial specification as a bit mask.

---

**FORMAT**            **LIB\$CRC\_TABLE** *polynomial-coefficient ,crc-table*

---

**RETURNS**            None.

---

**ARGUMENTS**        ***polynomial-coefficient***  
 VMS usage: **mask\_longword**  
 type:            **longword (unsigned)**  
 access:         **read only**  
 mechanism:      **by reference**

A bit mask indicating which polynomial coefficients are to be generated by LIB\$CRC\_TABLE. The **polynomial-coefficient** argument is the address of an unsigned longword integer containing this bit mask.

***crc-table***  
 VMS usage: **vector\_longword\_signed**  
 type:            **longword integer (signed)**  
 access:         **write only**  
 mechanism:      **by reference, array reference**

The 16-longword table that LIB\$CRC\_TABLE produces. The **crc-table** argument is the address of a signed longword integer containing the table.

---

**DESCRIPTION**      The table created by LIB\$CRC\_TABLE can be passed to the LIB\$CRC routine for generating the cyclic redundancy check value for a stream of characters.

For a description of how LIB\$CRC\_TABLE actually generates the table, see the *VAX Architecture Reference Manual*.

---

**CONDITION  
VALUES  
RETURNED**            *None.*

# LIB\$CRC\_TABLE

---

## EXAMPLE

```
1  %TITLE "Demonstrate LIB$CRC and LIB$CRC_TABLE"
   %SBTTL "Declarations"
   %IDENT "1-001"

!+
! FACILITY:
!
!   VMS Run-time library
!
! FUNCTIONAL DESCRIPTION:
!
!   This program demonstrates the use of LIB$CRC and LIB$CRC_TABLE.
!
! IMPLICIT INPUTS:
!
!   The user is requested to enter two strings.
!
! IMPLICIT OUTPUTS:
!
!   Output is printed to the controlling terminal.
!
! SIDE EFFECTS:
!
!   None
!
! AUTHOR:
!
!   Ken Cowan
!
! CREATION DATE: 22-Feb-1985
!
! MODIFICATION HISTORY:
!
!--

OPTION TYPE = EXPLICIT

DECLARE LONG      CRC_TABLE(15),      ! CRC table array &
                LONG      CRC_VAL_1,  ! CRC for first stream &
                LONG      CRC_VAL_2,  ! CRC for second stream &
                STRING     DATA_1,    ! First data stream &
                STRING     DATA_2     ! Second data stream

EXTERNAL LONG FUNCTION LIB$CRC          ! Rtn to calculate CRC
EXTERNAL SUB LIB$CRC_TABLE              ! Rtn to set up table for CRC

OPEN "SYS$INPUT:" FOR INPUT AS FILE 1%

!+
! Initialize the CRC table. Use the CRC-16 polynomial (refer to
! "VAX Architecture Handbook"). This is the polynomial used by
! DDCMP and Bisync.
!-

CALL LIB$CRC_TABLE( 0'120001'L, CRC_TABLE() BY REF )

!+
! Get data from user.
!-

LINPUT #1%, 'Enter string: ';DATA_1
```

# LIB\$CRC\_TABLE

```
!+
! Calc the CRC for the user's input. This CRC polynomial needs
! an initial CRC of 0 (refer to "VAX Architecture Handbook").
! LIB$CRC returns a longword, but only the low order word is valid
! for this polynomial.
!-

CRC_VAL_1 = LIB$CRC( CRC_TABLE() BY REF, 0%, DATA_1 )
CRC_VAL_1 = CRC_VAL_1 AND 32767%

!+
! Get more data from user.
!-

LINPUT #1%, 'Enter a second string: ';DATA_2

CRC_VAL_2 = LIB$CRC( CRC_TABLE() BY REF, 0%, DATA_2 )
CRC_VAL_2 = CRC_VAL_2 AND 32767%

!+
! Tell the user the results of the CRC comparison.
!-

IF CRC_VAL_1 = CRC_VAL_2
THEN
    PRINT "The two CRCs";CRC_VAL_1;" and ";CRC_VAL_2;" were the same"
ELSE
    PRINT "The two CRCs";CRC_VAL_1;" and ";CRC_VAL_2;" were the different"
END IF

IF DATA_1 = DATA_2
THEN
    PRINT "The two strings were the same"
ELSE
    PRINT "The two strings were different"
END IF

END
```

This BASIC example program shows the use of LIB\$CRC and LIB\$CRC\_TABLE. One example of the output generated by this program is as follows:

```
$ RUN CRC
Enter string: DOVE
Enter a second string: HOSE
The two CRCs 29915 and 29915 were the same
The two strings were different
```

# LIB\$CREATE\_DIR

---

## LIB\$CREATE\_DIR Create a Directory

The Create a Directory routine creates a directory or subdirectory.

---

<b>FORMAT</b>	<b>LIB\$CREATE_DIR</b> <i>device-directory-spec</i> [, <i>owner-UIC</i> ] [, <i>protection-enable</i> ] [, <i>protection-value</i> ] [, <i>maximum-versions</i> ] [, <i>relative-volume-number</i> ]
---------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>device-directory-spec</i></b> VMS usage: <b>device_name</b> type: <b>character string</b> access: <b>read only</b> mechanism: <b>by descriptor</b>
------------------	--

Directory specification of the directory or subdirectory that LIB\$CREATE\_DIR will create. The **device-directory-spec** argument is the address of a descriptor pointing to this directory specification.

The format of the **device-directory-spec** string conforms to standard Record Management Services (RMS) format. This specification must contain a directory or subdirectory specification. It may contain a disk specification. SMD\$:[THIS.IS.IT] is an example of a standard RMS file specification, where SMD\$ is the disk specification and [THIS.IS.IT] is the subdirectory specification.

This specification cannot contain a node name, file name, file type, file version, or wildcard characters. The maximum size of this string is 255 characters.

### ***owner-UIC***

VMS usage:	<b>uic</b>
type:	<b>longword (unsigned)</b>
access:	<b>read only</b>
mechanism:	<b>by reference</b>

User Identification Code (UIC) identifying the owner of the created directory or subdirectory. The **owner-UIC** argument is the address of an unsigned longword that contains the UIC. If **owner-UIC** is zero, the owner UIC is that of the parent directory.

This is an optional argument. The default is the UIC of the parent directory except when the directory is in UIC format. For a directory in UIC format, for example [123,321], the UIC of the created directory is used.

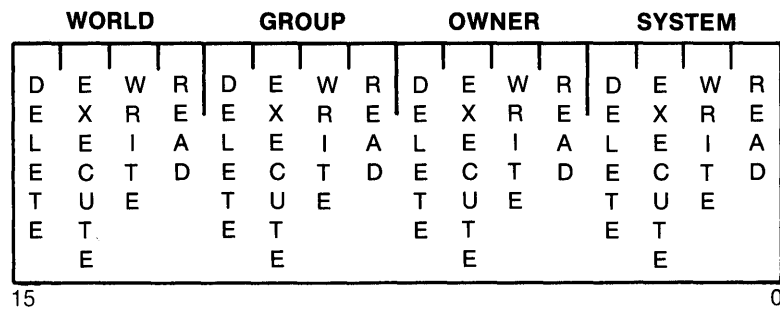
**protection-enable**

VMS usage: **mask\_word**  
 type: **word (unsigned)**  
 access: **read only**  
 mechanism: **by reference**

Mask specifying the bits of **protection-value** to be set. The **protection-enable** argument is the address of an unsigned word containing this protection mask.

Figure LIB-1 shows the structure of a protection mask. Access is allowed for bits set to zero.

**Figure LIB-1 Structure of a Protection Mask**



ZK-1979-84

Set bits in the **protection-enable** mask cause corresponding bits of **protection-value** to be set. Clear bits in the **protection-enable** mask cause corresponding bits of **protection-value** to take the value of the corresponding bit in the parent directory's file protection. Bits in the parent directory's file protection which indicate delete access will not cause corresponding bits of **protection-value** to be set, however.

Following is an example of how the **protection-value** protection mask is defined.

Mask Name	Hex Number	Value
Protection enable	%XDBFF	S:NONE, O:NONE, G:E, W:W
Parent directory	%X13FF	S:RWED, O:RWED, G:RW, W:R
Protection value	%X37FF	S:RWE, O:RWE, G:RWE, W:RW

**Protection-enable** is an optional argument. It should be used only when you wish to change protection values from the parent directory's default file protection. The default for **protection-enable** is a mask of all zero bits, which results in the propagation of the parent directory's file protection. If the **protection-enable** mask contains zeros, **protection-value** is ignored.



# LIB\$CREATE\_DIR

## ***protection-value***

VMS usage: **file\_protection**  
type: **word (unsigned)**  
access: **read only**  
mechanism: **by reference**

System/Owner/Group/World protection value of the directory you are creating. The **protection-value** argument is the address of an unsigned word which contains this protection mask.

The bits of **protection-value** are set or cleared in the method described in the definition of **protection-enable** above.

**Protection-value** is an optional argument. The default is a word of all zero bits, which specifies full access for all access categories. Typically, **protection-value** is not omitted unless **protection-enable** is also omitted. If **protection-enable** is omitted, **protection-value** will be ignored.

## ***maximum-versions***

VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **read only**  
mechanism: **by reference**

Maximum number of versions allowed for files created in the newly created directories. The **maximum-versions** argument is the address of an unsigned word containing the value of the maximum number of versions.

**Maximum-versions** is an optional argument. The default is the parent directory's default version limit. If specified as zero, the maximum number of versions is not limited.

## ***relative-volume-number***

VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **read only**  
mechanism: **by reference**

Relative volume number within a volume set on which the directory or subdirectory is created. The **relative-volume-number** argument is the address of an unsigned word containing the relative volume number. The **relative-volume-number** argument is optional. The default is arbitrary placement within the volume set.

---

## **DESCRIPTION**

LIB\$CREATE\_DIR allows the caller to indicate the owner and protection of the created directory or subdirectory. The caller can also indicate the maximum number of versions of a file which will be maintained and the relative volume number in which the directory or subdirectory will be created.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_CREATED	Routine successfully completed; one or more directories created.
SS\$_NORMAL	Routine successfully completed; all specified directories already exist.
LIB\$_INVARG	Invalid argument to Run_Time Library. Either the required argument was omitted, or <b>device-directory-spec</b> is longer than 255 characters.
LIB\$_INVFILSPE	Invalid file specification. Either the file specification did not contain an explicit directory and device name, or it contained a node name, file name, file type, file version, or wildcard. This error is also produced if the device specified was not a disk.

Any condition values returned by \$ASSIGN.

Any condition values returned by \$DASSGN.

Any condition values returned by \$PARSE.

Any condition values returned by \$QIO.

Any condition values returned by LIB\$ANALYZE\_SDESC.

Any condition values returned by LIB\$GET\_EF.

# LIB\$CREATE\_USER\_VM\_ZONE

---

## LIB\$CREATE\_USER\_VM\_ZONE      **Create User-Defined Storage Zone**

The Create User-Defined Storage Zone routine creates a new user-defined storage zone.

---

**FORMAT**                      **LIB\$CREATE\_USER\_VM\_ZONE**  
*zone-id* [,*user-argument*]  
                                  [,*user-allocation-procedure*]  
                                  [,*user-deallocation-procedure*]  
                                  [,*user-reset-procedure*]  
                                  [,*user-delete-procedure*] [,*zone-name*]

---

**RETURNS**                    VMS usage: **cond\_value**  
                                  type:        **longword (unsigned)**  
                                  access:     **write only**  
                                  mechanism: **by value**

---

**ARGUMENTS**                ***zone-id***  
VMS usage: **identifier**  
type:        **longword (unsigned)**  
access:      **write only**  
mechanism: **by reference**

Zone identifier. The **zone-id** argument is the address of a longword that receives the identifier of the newly created zone.

***user-argument***

VMS usage: **user\_arg**  
type:        **longword (unsigned)**  
access:      **read only**  
mechanism: **by reference**

User argument. The **user-argument** argument is the address of an unsigned longword containing the user argument. LIB\$CREATE\_USER\_VM\_ZONE copies the value of **user-argument** and supplies the value to all user-procedures invoked.

***user-allocation-procedure***

VMS usage: **procedure**  
type:        **procedure entry mask**  
access:      **function call (before return)**  
mechanism: **by value**

User allocation routine. The **user-allocation-procedure** argument is the address of the procedure entry mask for the optional user routine that is invoked each time LIB\$GET\_VM is called for the zone.

# LIB\$CREATE\_USER\_VM\_ZONE

## ***user-deallocation-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User deallocation routine. The **user-deallocation-procedure** argument is the address of the procedure entry mask for the optional user routine that is invoked each time LIB\$FREE\_VM is called for the zone.

## ***user-reset-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User routine to reset the zone. The **user-reset-procedure** argument is an optional user routine that is invoked each time LIB\$RESET\_VM\_ZONE is called for the zone.

## ***user-delete-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User routine to delete the zone. The **user-delete-procedure** argument is the address of the procedure entry mask for the optional user routine that is invoked when LIB\$DELETE\_VM\_ZONE is called for the zone.

## ***zone-name***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Name to be associated with the zone being created. The optional **zone-name** argument is the address of a descriptor pointing to the zone name. If **zone-name** is not specified, the zone will not have an associated name.

---

## **DESCRIPTION**

LIB\$CREATE\_USER\_VM\_ZONE creates a user-defined zone. If an error status is returned, the zone is not created.

Each time that one of the heap Management Routines (LIB\$GET\_VM, LIB\$FREE\_VM, LIB\$RESET\_VM\_ZONE, or LIB\$DELETE\_VM\_ZONE) is called to perform an operation on a user-defined zone, the corresponding user routine that you supplied is used.

You may omit any of the optional user routines. However, if you omit a routine and later call the corresponding heap management routine, the error status LIB\$\_INVOPEZON will be returned.

# LIB\$CREATE\_USER\_VM\_ZONE

## Call Format for User Routines

The user routines are called with arguments similar to those passed to LIB\$GET\_VM, LIB\$FREE\_VM, LIB\$RESET\_VM\_ZONE, or LIB\$DELETE\_VM\_ZONE. In each case, the **user-argument** argument from LIB\$CREATE\_USER\_VM\_ZONE is passed to the user routine rather than a **zone-id** argument.

The call format for a user get or free routine is as follows:

**user-rtn** num-bytes ,base-adr ,user-argument

### num-bytes

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Number of contiguous bytes to allocate or free. The **num-bytes** argument is the address of a longword integer containing the number of bytes. The value of **num-bytes** must be greater than zero.

### base-adr

VMS usage: **address**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

Virtual address of the first contiguous block of bytes allocated or freed. The **base-adr** argument is the address of an unsigned longword containing this base address. (This argument is write-only for a get routine, and read-only for a free routine.)

### user-argument

VMS usage: **user\_arg**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

User argument. LIB\$CREATE\_USER\_VM\_ZONE copies **user-argument** as it is supplied to all user routines invoked.

The status value returned by your routine is returned as the status value for the corresponding call to LIB\$GET\_VM or LIB\$FREE\_VM.

The **zone-id** value that is returned can be used in calls to LIB\$SHOW\_VM\_ZONE and LIB\$VERIFY\_VM\_ZONE.

The call format for a user reset or delete routine is as follows:

**user-rtn** user-argument

### user-argument

VMS usage: **user\_arg**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

User argument. LIB\$CREATE\_USER\_VM\_ZONE copies **user-argument** as it is supplied to all user routines invoked.

# LIB\$CREATE\_USER\_VM\_ZONE

The status value returned by your routine is returned as the status value for the corresponding call to LIB\$RESET\_VM\_ZONE or LIB\$DELETE\_VM\_ZONE.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_INSVIRMEM	Insufficient virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor for <b>zone-name</b> .

# LIB\$CREATE\_VM\_ZONE

---

## LIB\$CREATE\_VM\_ZONE Create a New Zone

The Create a New Zone routine creates a new storage zone according to specified arguments.

---

**FORMAT**            **LIB\$CREATE\_VM\_ZONE**    *zone-id* [, *algorithm*]  
  [, *algorithm-argument*]  
  [, *flags*] [, *extend-size*]  
  [, *initial-size*] [, *block-size*]  
  [, *alignment*] [, *page-limit*]  
  [, *smallest-block-size*]  
  [, *zone-name*]  
  [, *number-of-areas*]  
  [, *get-page*] [, *free-page*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        ***zone-id***  
                          VMS usage: **identifier**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by reference**

Zone identifier. The ***zone-id*** argument is the address of a longword that is set to the zone identifier of the newly created zone.

### ***algorithm***

VMS usage: **longword\_signed**  
type:        **longword integer (signed)**  
access:     **read only**  
mechanism: **by reference**

Algorithm. The ***algorithm*** argument is the address of a longword integer that represents the code for one of the LIB\$VM algorithms:

- |   |                     |                                |
|---|---------------------|--------------------------------|
| 1 | LIB\$_VM_FIRST_FIT  | First fit                      |
| 2 | LIB\$_VM_QUICK_FIT  | Quick fit, lookaside list      |
| 3 | LIB\$_VM_FREQ_SIZES | Frequent sizes, lookaside list |
| 4 | LIB\$_VM_FIXED      | Fixed size blocks              |

If ***algorithm*** is not specified, a default of 1 (first fit) is used.

# LIB\$CREATE\_VM\_ZONE

## *algorithm-argument*

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Algorithm argument. The **algorithm-argument** argument is the address of a longword integer that contains a value specific to the particular allocation algorithm.

Algorithm	Value
First fit	Not used, may be omitted.
Quick fit	The number of lookaside lists used. The number of lists must be between 1 and 128.
Frequent sizes	The number of lookaside lists used. The number of lists must be between 1 and 16.
Fixed size blocks	The fixed request size (in bytes) for each get or free request. The request size must be greater than 0.

The **algorithm-argument** argument must be specified if you are using the quick-fit, frequent-sizes or fixed-size-blocks algorithms. However, this argument is optional if you are using the first-fit algorithm.

## *flags*

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Flags. The **flags** argument is the address of a longword integer that contains flag bits that control various options.

Bit	Value	Description
0	LIB\$_VM_BOUNDARY_TAGS	Boundary tags for faster freeing Adds a minimum of eight bytes to each block
1	LIB\$_VM_GET_FILL0	LIB\$GET_VM; fill with bytes of 0
2	LIB\$_VM_GET_FILL1	LIB\$GET_VM; fill with bytes of FF (hexadecimal)
3	LIB\$_VM_FREE_FILL0	LIB\$FREE_VM; fill with bytes of 0
4	LIB\$_VM_FREE_FILL1	LIB\$FREE_VM; fill with bytes of FF (hexadecimal)
5	LIB\$_VM_EXTEND_AREA	Add extents to existing areas if possible

Bits 6 through 31 are reserved and must be 0.

This is an optional argument. If **flags** is omitted, the default of 0 (no fill and no boundary tags) is used.



# LIB\$CREATE\_VM\_ZONE

## ***extend-size***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Zone extend size. The **extend-size** argument is the address of a longword integer that contains the number of (512-byte) pages to be added to the zone each time it is extended.

The value of **extend-size** must be between 1 and 1024.

This is an optional argument. If **extend-size** is not specified, a default of 16 pages is used.

**Note:** *Extend-size* does not limit the number of blocks that can be allocated from the zone. The actual extension size is the greater of **extend-size** and the number of pages needed to satisfy the LIB\$GET\_VM call that caused the extension.

## ***initial-size***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Initial size for the zone. The **initial-size** argument is the address of a longword integer that contains the number of (512-byte) pages to be allocated for the zone as the zone is created.

This is an optional argument. If **initial-size** is not specified or is specified as 0, no pages are allocated when the zone is created. The first call to LIB\$GET\_VM for the zone allocates **extend-size** pages.

## ***block-size***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Block size of the zone. The **block-size** argument is the address of a longword integer specifying the allocation quantum (in bytes) for the zone. All blocks allocated are rounded up to a multiple of **block-size**.

The value of **block-size** must be a power of 2 between 8 and 512. This is an optional argument. If **block-size** is not specified, a default of 8 is used.

## ***alignment***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Block alignment. The **alignment** argument is the address of a longword integer that specifies the required address alignment (in bytes) for each block allocated.

The value of **alignment** must be a power of 2 between 4 and 512. This is an optional argument. If **alignment** is not specified, a default of 8 (quadword alignment) is used.

# LIB\$CREATE\_VM\_ZONE

## ***page-limit***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Maximum page limit. The **page-limit** argument is the address of a longword integer that specifies the maximum number of (512-byte) pages that can be allocated for the zone. The value of **page-limit** must be between 0 and 32,767. Note that part of the zone is used for header information.

This is an optional argument. If **page-limit** is not specified or is specified as 0, the only limit is the total process virtual address space limit imposed by VMS. If **page-limit** is specified, then **initial-size** must also be specified.

## ***smallest-block-size***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Smallest block size. The **smallest-block-size** argument is the address of a longword integer that specifies the smallest block size (in bytes) that has a lookaside list for the quick fit algorithm.

If **smallest-block-size** is not specified, the default of **block-size** is used. That is, lookaside lists are provided for the first *n* multiples of **block-size**.

## ***zone-name***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Name to be associated with the zone being created. The optional **zone-name** argument is the address of a descriptor pointing to the zone name. If **zone-name** is not specified, the zone will not have an associated name.

## ***number-of-areas***

VMS usage: **longword\_signed**  
type: **longword (signed)**  
access: **read only**  
mechanism: **by reference**

Number of areas into which the memory should be subdivided. The **number-of-areas** argument is the address of a longword integer containing the number of subdivided memory areas.

## ***get-page***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **read only**  
mechanism: **by value**

Routine that allocates pages of memory. The **get-page** argument is the address of a procedure entry mask used to allocate pages of memory.

# LIB\$CREATE\_VM\_ZONE

## *free-page*

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **read only**  
mechanism: **by value**

Routine that deallocates pages of memory. The **free-page** argument is the address of a procedure entry mask used to deallocate pages of memory.

---

## DESCRIPTION

LIB\$CREATE\_VM\_ZONE creates a new storage zone. The zone identifier value that is returned can be used in calls to LIB\$GET\_VM, LIB\$FREE\_VM, LIB\$RESET\_VM\_ZONE, LIB\$DELETE\_VM\_ZONE, LIB\$SHOW\_VM\_ZONE, LIB\$VERIFY\_VM\_ZONE, and LIB\$CREATE\_USER\_VM\_ZONE.

The following restrictions apply when you are creating a zone.

- If you want the zone to be accessible from another process or processes, you must map the global section into the same virtual addresses in all processes. You can use PPL\$CREATE\_SHARED\_MEM to map to a global section, after you have first called PPL\$INITIALIZE.
- The zone cannot expand; in other words, additional areas cannot be added to the zone.
- The restrictions for LIB\$RESET\_VM\_ZONE also apply to shared zones. That is, it is the caller's responsibility to ensure that the caller has exclusive access to the zone while the reset operation is being performed.

If an error status is returned, the zone is not created.

# LIB\$CREATE\_VM\_ZONE

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_INSVIRMEM	Insufficient virtual memory.
LIB\$_INVARG	Invalid argument.
LIB\$_INVSTRDES	Invalid string descriptor for <b>zone-name</b> .



## **flags**

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Value used in selecting the contents of the KEY2 and VAL2 fields; **flags** is stored with the entry. The **flags** argument is the address of an unsigned longword containing the flags. When preparing the output line, LIB\$CRF\_OUTPUT uses **flags** and the 16-bit mask in the field descriptor table to extract the data. The high-order bit of the word is reserved for LIB\$CRF\_INS\_KEY.

---

## **DESCRIPTION**

LIB\$CRF\_INS\_KEY stores information to be printed in the KEY1, KEY2, VAL1, and VAL2 fields. When you call this routine, an entry for the key is made in the cross-reference table if the key is not present in the table. If the key is present, only the value address and value flag fields are updated.

Using LIB\$CRF\_INS\_KEY involves the following steps:

- Define a table of control information, using the \$CRFCTLTABLE macro.
- Define each field of the output line, using the \$CRFFIELD macro.
- Specify the end of each set of macros that define a field in the output line, using the \$CRFFIELDEND macro.
- Provide data by calling LIB\$CRF\_INS\_KEY to insert an entry for the specify key in the specified symbol table. This data is used to build tables in virtual memory.
- Call LIB\$CRF\_OUTPUT, the cross-reference output routine, to summarize and format the data. Supply a routine that LIB\$CRF\_OUTPUT calls to print each line in the output file. Because you supply this routine, you can control the number of lines per page and the header lines.

---

## **CONDITION VALUES RETURNED**

*None.*

# LIB\$CRF\_INS\_REF

---

## LIB\$CRF\_INS\_REF Insert Reference to a Key in the Cross-Reference Table

The Insert Reference to a Key in the Cross-Reference Table routine inserts a reference to a key in a cross-reference symbol table.

---

<b>FORMAT</b>	<b>LIB\$CRF_INS_REF</b> <i>control-table ,longword-integer-key ,reference-string ,longword-integer-reference ,ref-definition-indicator</i>
---------------	--

---

<b>RETURNS</b>	None.
----------------	-------

---

### ARGUMENTS

#### ***control-table***

VMS usage: **vector\_longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference, array reference**

Control table associated with this cross-reference. The **control-table** argument is the address of an array containing the control table.

#### ***longword-integer-key***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Key referred to by LIB\$CRF\_INS\_REF. The **longword-integer-key** argument is the address of a signed longword integer containing the key. The key is a counted ASCII string that contains a symbol name or an unsigned binary longword. It must be a permanent address in the user's symbol table.

#### ***reference-string***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Counted ASCII string with a maximum of 31 characters, not including the byte count. The **reference-string** argument is the address of a descriptor pointing to the counted ASCII string.

***longword-integer-reference***

VMS usage: **longword\_signed**  
 type: **longword integer (signed)**  
 access: **write only**  
 mechanism: **by reference**

The 16-bit value used in selecting the contents of the REF1 field. The **longword-integer-reference** argument is the address of a signed longword integer containing this value. When preparing the output line, LIB\$CRF\_OUTPUT uses **longword-integer-reference** and the bit mask in the field descriptor table to extract the data. The high-order bit of the word is reserved for LIB\$CRF\_INS\_REF.

***ref-definition-indicator***

VMS usage: **longword\_signed**  
 type: **longword integer (signed)**  
 access: **read only**  
 mechanism: **by reference**

Reference/definition indicator that LIB\$CRF\_INS\_REF uses to distinguish between a reference to a symbol and the definition of the symbol. The **ref-definition-indicator** argument is the address of a signed longword integer containing this indicator. The only difference between processing a symbol reference and a symbol definition is where LIB\$CRF\_INS\_REF stores the information.

The reference/definition indicator can have either of the following values:

Symbolic Name	Description
CRF\$K_REF	Reference to a symbol
CRF\$K_DEF	Definition of a symbol

**DESCRIPTION**

LIB\$CRF\_INS\_REF inserts a reference to a key in the cross-reference symbol table. If you attempt to insert reference information for a key that was not specified in a call to LIB\$CRF\_INS\_KEY, LIB\$CRF\_INS\_REF uses the address of the key to locate the symbol name and set the KEY1 field. Once set, either as a result of LIB\$CRF\_INS\_KEY or LIB\$CRF\_INS\_REF, the KEY1 field is never changed. A KEY1 field set by LIB\$CRF\_INS\_REF has a space-filled VAL1 field associated with it unless it is overridden by a subsequent call to LIB\$CRF\_INS\_KEY.

Using LIB\$CRF\_INS\_REF involves the following steps:

- 1 Define a table of control information, using the \$CRFCTLTABLE macro.
- 2 Define each field of the output line, using the \$CRFFIELD macro.
- 3 Specify the end of each set of macros that define a field in the output line, using the \$CRFFIELDEND macro.
- 4 Provide data by calling LIB\$CRF\_INS\_REF to insert a reference to a key in the specified symbol table. This data is used to build tables in virtual memory.



## LIB\$CRF\_INS\_REF

- 5 Call LIB\$CRF\_OUTPUT, the cross-reference output routine, to summarize and format the data. Supply a routine that LIB\$CRF\_OUTPUT calls to print each line in the output file. Because you supply this routine, you can control the number of lines per page and the header lines.

---

**CONDITION  
VALUES  
RETURNED**

*None.*

---

## LIB\$CRF\_OUTPUT Output Cross-Reference Table Information

The Output Cross-Reference Table Information routine extracts the information from the cross-reference tables and formats the output pages.

---

**FORMAT**            **LIB\$CRF\_OUTPUT**    *control-table ,output-line-width  
,page1 ,page2 ,mode-indicator  
,delete-save-indicator*

---

**RETURNS**            None.

---

### ARGUMENTS

#### ***control-table***

VMS usage: **vector\_longword\_signed**  
 type:            **longword integer (signed)**  
 access:         **read only**  
 mechanism:     **by reference, array reference**

Control table associated with the cross-reference. The **control-table** argument is the address of an array containing the control table. The table contains the address of the user-supplied routine that prints the lines formatted by LIB\$CRF\_OUTPUT.

#### ***output-line-width***

VMS usage: **longword\_signed**  
 type:            **longword integer (signed)**  
 access:         **read only**  
 mechanism:     **by reference**

Width of the output line. The **output-line-width** argument is the address of a signed longword integer containing the width.

#### ***page1***

VMS usage: **longword\_signed**  
 type:            **longword integer (signed)**  
 access:         **read only**  
 mechanism:     **by reference**

Number of lines on the first page of the output. The **page1** argument is the address of a signed longword integer containing this number. This allows the user to reserve space to print header information on the first page of the cross-reference.

#### ***page2***

VMS usage: **longword\_signed**  
 type:            **longword integer (signed)**  
 access:         **read only**  
 mechanism:     **by reference**

# LIB\$CRF\_OUTPUT

Number of lines per page for the other pages. The **page2** argument is the address of a signed longword integer containing this number.

## ***mode-indicator***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Output mode indicator. The **mode-indicator** argument is the address of a signed longword integer containing the mode indicator.

This indicator allows the user to select which of three output modes is desired.

<b>Output Mode</b>	<b>Description</b>
CRF\$_VALUES	Only the value and key fields are to be printed. LIB\$CRF_OUTPUT creates multiple columns across the page. Each column consists of the KEY1, KEY2, VAL1, and VAL2 fields. A minimum of one space between each column is guaranteed.
CRF\$_VALS_REFS	Requests a cross-reference summary that has no column space saved for a defining reference. If the user inserted a reference with the CRF\$_DEF indicator, the entry is ignored.
CRF\$_DEFS_REFS	Requests a cross-reference summary with the first REF1 and REF2 fields used only for definition references. If no definition reference is provided, the fields are space filled.

## ***delete-save-indicator***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Delete/save indicator which LIB\$CRF\_OUTPUT uses to determine whether the table's built-in accumulating symbol information is to be saved or deleted once the cross-reference is produced. The **delete-save-indicator** argument is the address of a signed longword integer containing the delete/save indicator.

The indicator can be either of the following:

CRF\$\_SAVE        To preserve the tables for subsequent processing  
CRF\$\_DELETE     To delete the tables

---

## **DESCRIPTION**

LIB\$CRF\_OUTPUT can format output lines for three types of cross-reference listings.

- 1** A summary of symbol names and their values, as illustrated in Figure LIB-2.
- 2** A summary of symbol names, their values, and the names of modules that refer to the symbol, as illustrated in Figure LIB-3.

- 3 A summary of symbol names, their values, the name of the defining module, and the names of those modules that refer to the symbol, as Figure LIB-4 shows.

**Figure LIB-2 Summary of Symbol Names and Values**

```

+-----+
! Symbols By Name !
+-----+

Symbol          Value          Symbol          Value
-----
BAS$INSTR       000020B0-RU   BAS$SCRATCH     00002308-RU
BAS$IN_D_R      000021F0-RU   BAS$STATUS      00002338-RU
BAS$IN_F_R      000021E8-RU   BAS$STR_D       000020C0-RU
BAS$IN_L_R      000021E0-RU   BAS$STR_F       000020B8-RU
BAS$IN_T_DX     000021F8-RU   BAS$STR_L       000020C8-RU
BAS$IN_W_R      000021D8-RU   BAS$UNLOCK      00002310-RU
BAS$ID_END      000021D0-RU   BAS$UPDATE      000022E8-RU
BAS$LINKAGE     00001674-R    BAS$UPDATE_COUN 000022F0-RU
BAS$LINPUT      000021A8-RU   BAS$VAL_D       00002110-RU
BAS$MAT_INPUT   00002268-RU   BAS$VAL_F       00002108-RU

```

ZK-1973-84

**Figure LIB-3 Summary of Symbol Names, Values, and Name of Referring Modules**

```

Symbol          Value          Referenced By ...
-----
BAS$K_DIVBY_ZER 0000003D      ALLGBL          BAS$ERROR
                  BAS$POWDJ      BAS$POWII
                  BAS$POWRJ      BAS$POWRR
BAS$K_DUPKEYDET 00000086      ALLGBL          BAS$$SIGNAL_IO
BAS$K_ENDFILDEV 0000000B      ALLGBL          BAS$$REC_PROC
                  BAS$$UDF_RL
BAS$K_ENDOF_STA 0000006C      ALLGBL

```

ZK-1974-84

# LIB\$CRF\_OUTPUT

**Figure LIB-4 Summary Indicating Defining Module**

Symbol	Value	Defined By	Referenced By ...
LIB\$FREE_VM	0001E185-R	LIB\$VM	ALLGBL BAS\$MARGIN BAS\$XLATE FOR\$VM STR\$APPEND STR\$DUPL_CHAR STR\$REPLACE
LIB\$GET_COMMAND	0001E2B0-R	LIB\$GET_INPUT	ALLGBL
LIB\$GET_COMMON	0001E4D6-R	LIB\$COMMON	ALLGBL

ZK-1971-84

Regardless of the format of the output, LIB\$CRF\_OUTPUT considers the output line as consisting of six different field types.

KEY1	Is the first field in the line. It contains a symbol name.
KEY2	Is the second field in the line. It contains a set of flags (for example, -R) that provide information about the symbol.
VAL1	Is the third field in the line. It contains the value of the symbol.
VAL2	Is the fourth field in the line. It contains a set of flags describing VAL1.
REF1 and REF2 fields	Within each REF1 and REF2 pair, REF1 provides a set of flags and REF2 provides the name of a module that references the symbol.

Any of these fields can be omitted from the output.

For example:

Symbol	Value	Symbol	Value
BAS\$INSTR	000020B0-RU	BAS\$SCRATCH	00002308-RU
KEY1	VAL1 VAL2	KEY1	VAL1 VAL2
Symbol	Value	Defined By	Referenced By ...
LIB\$FREE_VM	0001E185-R	LIB\$VM	ALLGBL
KEY1	VAL1 VAL2	REF2 (CRF\$K_DEF)	REF2 (CRF\$K_REF)

**CONDITION  
VALUES  
RETURNED**

*None.*

---

## LIB\$CURRENCY Get System Currency Symbol

The Get System Currency Symbol routine returns the system's currency symbol.

---

**FORMAT**            **LIB\$CURRENCY**    *currency-string* [, *resultant-length*]

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        ***currency-string***  
                           VMS usage: **char\_string**  
                           type:            **character string**  
                           access:        **write only**  
                           mechanism:    **by descriptor**

Currency symbol. The ***currency-string*** argument is the address of a descriptor pointing to the currency symbol.

***resultant-length***  
 VMS usage: **word\_unsigned**  
 type:            **word (unsigned)**  
 access:        **write only**  
 mechanism:    **by reference**

Number of characters that LIB\$CURRENCY has written into the ***currency-string*** argument, not counting padding in the case of a fixed-length string. The ***resultant-length*** argument is the address of an unsigned word containing the length of the currency symbol. If the input string is truncated to the size specified in the ***currency-string*** argument, ***resultant-length*** is set to this size. Therefore, ***resultant-length*** can always be used by the calling program to access a valid substring of ***currency-string***.

---

**DESCRIPTION**      LIB\$CURRENCY attempts to translate the logical name SYS\$CURRENCY as a process, group, or system logical name, in that order. If the translation fails, the routine returns the United States currency symbol (\$). If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS\$CURRENCY as a system-wide logical name to provide a default for all users, and an individual user with a special need can define SYS\$CURRENCY as a process logical name to override the system default.

For example, if you wish to use the British pound sign as the currency symbol within your process, but wish to leave the dollar sign as the system's default, define SYS\$CURRENCY to be the pound sign (#) in your process logical name table. After this, any call to LIB\$CURRENCY within your process returns the pound sign (#), while any call outside your process returns the dollar sign (\$).

# LIB\$CURRENCY

LIB\$CURRENCY is implicitly used by BASIC.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Successfully completed, but the currency string was truncated.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.

---

## EXAMPLE

```
10 !+
   ! This BASIC program uses LIB$CURRENCY to
   ! return the default system currency symbol.
   !-
   OUTLEN = 1
   CALL LIB$CURRENCY (CURR$, OUTLEN)
   PRINT CURR$
99 END
```

The BASIC program listed above uses LIB\$CURRENCY to display the system currency symbol default. The output generated by the program is a dollar sign (\$).





# LIB\$CVT\_DX\_DX

**word-integer-dest-length** argument contains the address of an unsigned word containing this length.

If the destination string is truncated, the returned length reflects the truncation. This word can be used by the calling program to determine if truncation has occurred or to extract the exact length of the string when the string contains space filling.

---

## DESCRIPTION

LIB\$CVT\_DX\_DX is a universal conversion utility routine. Although some of the functions of this routine may be found in other Run-Time Library routines, LIB\$CVT\_DX\_DX packages the conversion functions with a general interface. Because of this general interface, the calling program does not have to specify what conversion should be done for which data type.

The description of this routine has been divided into the following parts:

- Guidelines for Using LIB\$CVT\_DX\_DX
- Use of Numeric Byte Data Strings

### Guidelines for Using LIB\$CVT\_DX\_DX

The data type and descriptor class of the source and destination arguments determine how LIB\$CVT\_DX\_DX performs the conversion, according to the rules listed below.

- Conversion is defined over three sets of data types: atomic, string, and numeric byte data strings (NBDS). (For more information about numeric byte data strings, see the next section "Use of Numeric Byte Data Strings.") Although the set of data types in NBDS is actually a subset of the atomic and string data types, the three sets are mutually exclusive in this routine.
- Scale is applied when indicated in the descriptor (DSC\$K\_CLASS\_SD only) and scaling is defined for the data type.
- No language-specific semantics are applied, such as BASIC scale for DSC\$K\_DTYPE\_D.
- Some conversions must use intermediate values to arrive at the destination requested. Although some loss of speed is inevitable, intermediate values will not cause a loss of precision.
- Results are always rounded instead of truncated, except for the case described below. Note that loss of precision or range may be inherent in the destination data type or in the NBDS destination size. No errors are reported if there is a loss of precision or range as a result of destination data type.
- When the destination is an NBDS, and has fixed-string semantics, then if the source does not fill the destination, the destination is padded with blanks.
- When the source and destination are both NBDS, and no scaling is requested, then a straight copy is done without translation or conversion, and truncation is possible. If scaling is requested, then a conversion takes place as defined in Table LIB-3.
- When the source is an NBDS and the destination is non-NBDS, if there is an invalid character in the source, or the value is outside the range that can be represented by the destination, then LIB\$\_INVNBDS is returned.

- Attempts to convert a negative value to an unsigned data type cause the LIB\$\_INVCVT error to be returned.
- If the destination is an NBDS of class DSC\$K\_CLASS\_D, then a new string of appropriate size will be allocated for it if necessary.
- Invalid conversions resulting in an error produce an unpredictable result.

### Use of Numeric Byte Data Strings

For simplicity, and to define a generic numeric string that LIB\$CVT\_DX\_DX understands to be a numeric string, the set Numeric Byte Data String (NBDS) is defined to be the set of descriptors given in Table LIB-1.

**Table LIB-1 Acceptable Subset of VAX Standard Data Types**

DSC\$K_DTYPE_yyy	DSC\$K_CLASS_xxx					
	A	D	NCA	S	SD	VS
B						
BU	x		x			
T	x	x	x	x	x	x
VT						x

Note: An array will have the semantics of a fixed string. NBDS must have the format defined later.

ZK-4260-85

The combination of data type and descriptor class determines whether an argument is an NBDS. Table LIB-2 shows the combinations of descriptor class and data type (as specified in the fields of the descriptor) that LIB\$CVT\_DX\_DX recognizes. The combinations marked NBDS are considered NBDS's.

# LIB\$CVT\_DX\_DX

**Table LIB-2 Data Types Accepted by LIB\$CVT\_DX\_DX**

From:	To:
Text (decimal)	Longword
Text (hexadecimal)	Longword
Text (octal)	Longword
Text (binary)	Longword
Text (signed integer)	Longword
Text (logical)	Longword
Text (octal)	Longword
Text (hexadecimal)	Longword
Text	D__floating
Text	F__floating
Text	G__floating
Text	H__floating
Longword	Text (binary)
Longword	Text (signed integer)
Longword	Text (logical)
Longword	Text (octal)
Longword	Text (hexadecimal)

ZK-1942-84

For example, LIB\$CVT\_DX\_DX treats the combination DSC\$K\_DTYPE\_B/DSC\$K\_CLASS\_S (unsigned byte scalar) as an atomic data type. However, the routine considers DSC\$K\_DTYPE\_BU/DSC\$K\_CLASS\_NCA (noncontiguous array of unsigned bytes) to be an NBDS.

A destination NBDS is always left-justified.

If a destination NBDS requires more than 50 digits for its format (including the sign, if any), then it is expressed in exponential format.

For a conversion of NBDS to NBDS, this format is used if scaling is requested. Otherwise, a straight copy is done. The format of a source NBDS is the same as the format defined for the input argument **inp** in OTS\$\_CVT\_T\_z, with bits 0, 2, and 4 set in the **flags** argument. That is, blanks are ignored, underflow causes an error and tabs are ignored.

Table LIB-3 defines the format of a destination NBDS.

**Table LIB-3 Destination NBDS Formats**

Source Data Type	Destination NBDS Format <sup>1</sup>
Byte integer (signed)	sdigits
Byte (unsigned)	sdigits
Word integer (signed)	sdigits
Word (unsigned)	digits
Longword integer (signed)	sdigits
Longword (unsigned)	digits
Quadword integer (signed)	sdigits
D_floating	s0.min(16,w-7)E(+ or -)nn
F_floating	s0.min( 7,w-7)E(+ or -)nn
G_floating	s0.min(15,w-8)E(+ or -)nnn
H_floating	s0.min(33,w-9)E(+ or -)nnnn
NBDS	s0.min(33,w-9)E(+ or -)nnnn
Decimal string	sdigits (as defined by VAX architecture)

<sup>1</sup>digits—Digits 0 through 9, and a decimal point only if source descriptor specifies DSC\$B\_SCALE less than 0.

w—Width of destination in bytes.

s—Sign. For positive numbers the sign is implied.

min—Minimum of two values.

Two array descriptors, DSC\$K\_CLASS\_A and DSC\$K\_CLASS\_NCA, are supported for specific languages that describe strings using these mechanisms.

- DSC\$B\_DIMCT = 1—Only one-dimensional arrays are recognized.
- DSC\$W\_LENGTH = 1—The length of each array element must be a byte.
- DSC\$L\_ARSIZE less than or equal to 65,535—The total size of the array must be less than 65,535 bytes.
- If DSC\$L\_ARSIZE = 0, the array has a length of zero.
- DSC\$L\_S1 = 1—The stride of an array passed by a noncontiguous array descriptor must be 1. That is, even if the class of the array's descriptor is noncontiguous array (NCA), the array itself must be contiguous.
- An array is written with the semantics of a fixed string.

For more information about the semantics of writing output strings, see *VMS RTL String Manipulation (STR\$) Manual*.

# LIB\$CVT\_DX\_DX

If the calling program passes a descriptor to LIB\$CVT\_DX\_DX that does not comply with Table LIB-2, one of the following error messages is returned:

LIB\$\_INVDTYDSC  
LIB\$\_INVCLADSC  
LIB\$\_INVCLADTY  
LIB\$\_INVNBDS

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_DECOVF	Packed decimal overflow error. Severe error.
LIB\$_FLTOVF	Floating overflow error. Severe error.
LIB\$_FLTUND	Floating underflow error. Severe error.
LIB\$_INVCLADSC	Invalid class in descriptor. This class of descriptor is not supported. Severe error.
LIB\$_INVCLADTY	Invalid class and data type in descriptor. This class and data type combination is not supported. Severe error.
LIB\$_INVCVT	If the source value is negative and the destination data type is unsigned, this error is returned.
LIB\$_INVDTYDSC	Invalid data type in descriptor. This data type is not supported. Severe error.
LIB\$_INTOVF	Integer overflow error. Severe error.
LIB\$_INVNBDS	Invalid NBDS. There is an invalid character in the input, or the value is outside the range that can be represented by the destination, or the NMDS descriptor is invalid. This error is also signaled when the array size of an NBDS is larger than 65,535 bytes or the array is multi-dimensional.
LIB\$_OUTSTRTRU	Output string truncated. This is returned only when NBDS is both source and destination and no scaling is requested. The result is truncated.
LIB\$_ROPRAND	Reserved operand error. Severe Error.



# LIB\$CVT\_FROM\_INTERNAL\_TIME

Operation	Return Range	Type
LIB\$_MONTH_OF_YEAR	1 to 12	Absolute
LIB\$_DAY_OF_YEAR	1 to 366	Absolute
LIB\$_HOUR_OF_YEAR	1 to 8784	Absolute
LIB\$_MINUTE_OF_YEAR	1 to 527,040	Absolute
LIB\$_SECOND_OF_YEAR	1 to 31,622,400	Absolute
LIB\$_DAY_OF_MONTH	1 to 31	Absolute
LIB\$_HOUR_OF_MONTH	1 to 744	Absolute
LIB\$_MINUTE_OF_MONTH	1 to 44,640	Absolute
LIB\$_SECOND_OF_MONTH	1 to 2,678,400	Absolute
LIB\$_DAY_OF_WEEK	1 to 7	Absolute <sup>1</sup>
LIB\$_HOUR_OF_WEEK	1 to 168	Absolute <sup>2</sup>
LIB\$_MINUTE_OF_WEEK	1 to 10,080	Absolute <sup>3</sup>
LIB\$_SECOND_OF_WEEK	1 to 604,800	Absolute <sup>4</sup>
LIB\$_HOUR_OF_DAY	1 to 24	Absolute
LIB\$_MINUTE_OF_DAY	1 to 1440	Absolute
LIB\$_SECOND_OF_DAY	1 to 86,400	Absolute
LIB\$_MINUTE_OF_HOUR	1 to 60	Absolute
LIB\$_SECOND_OF_HOUR	1 to 3600	Absolute
LIB\$_SECOND_OF_MINUTE	1 to 60	Absolute
LIB\$_JULIAN_DATE	Julian date	Absolute <sup>5</sup>
LIB\$_DELTA_WEEKS		Delta <sup>6</sup>
LIB\$_DELTA_DAYS		Delta <sup>7</sup>
LIB\$_DELTA_HOURS		Delta <sup>8</sup>
LIB\$_DELTA_MINUTES		Delta <sup>9</sup>
LIB\$_DELTA_SECONDS		Delta <sup>10</sup>

<sup>1</sup>Day 1 is Monday

<sup>2</sup>Hours since midnight on previous Monday

<sup>3</sup>Minutes since midnight on previous Monday

<sup>4</sup>Seconds since midnight on previous Monday

<sup>5</sup>Number of days since system zero time (17-Nov-1858)

<sup>6</sup>Whole weeks

<sup>7</sup>Whole days

<sup>8</sup>Whole hours

<sup>9</sup>Whole minutes

<sup>10</sup>Whole seconds

# LIB\$CVT\_FROM\_INTERNAL\_TIME

## *resultant-time*

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by reference**

The external time that results from the conversion. The **resultant-time** argument is the address of an unsigned longword containing the result.

## *input-time*

VMS usage: **date\_time**  
type: **quadword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Optional absolute or delta time to be converted. The **input-time** argument is the address of an unsigned quadword containing the time. If you do not supply a value for **input-time**, the current system time is used.

---

## DESCRIPTION

LIB\$CVT\_FROM\_INTERNAL\_TIME converts an internal VMS system time (either absolute or delta) into an external time. The **operation** argument specifies the conversion. LIB\$CVT\_FROM\_INTERNAL\_TIME converts the value of **input-time** (or the current system time if **input-time** is not supplied) into one of the external formats listed in the **operation** argument description. LIB\$CVT\_FROM\_INTERNAL\_TIME then places the result into **resultant-time**.

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Normal successful completion.
LIB\$_IVTIME	Invalid time.
LIB\$_WRONUMARG	Incorrect number of arguments.
LIB\$_INVOPER	Invalid operation.
LIB\$_ABSTIMREQ	Absolute time required but delta time supplied.
LIB\$_DELTIMREQ	Delta time required but absolute time supplied.



# LIB\$CVTF\_FROM\_INTERNAL\_TIME

---

## LIB\$CVTF\_FROM\_INTERNAL\_TIME **Convert Internal Time to External Time (F-Floating Point Value)**

The Convert Internal Time to External Time (F-Floating Point Value) routine converts a delta internal VMS system time into an external F-floating time.

---

**FORMAT**                **LIB\$CVTF\_FROM\_INTERNAL\_TIME**    *operation*  
  *,resultant-time*  
  *,input-time*

---

**RETURNS**              VMS usage: **cond\_value**  
                              type:         **longword (unsigned)**  
                              access:        **write only**  
                              mechanism: **by value**

---

**ARGUMENTS**            **operation**  
                              VMS usage: **function\_code**  
                              type:         **longword (unsigned)**  
                              access:        **read only**  
                              mechanism: **by reference**

The conversion to be performed. The **operation** argument is the address of an unsigned longword specifying the operation. Valid values for **operation** are the following:

---

<b>Operation</b>	<b>Interpretation</b>
LIB\$_DELTA_WEEKS_F	Fractional weeks
LIB\$_DELTA_DAYS_F	Fractional days
LIB\$_DELTA_HOURS_F	Fractional hours
LIB\$_DELTA_MINUTES_F	Fractional minutes
LIB\$_DELTA_SECONDS_F	Fractional seconds

---

# LIB\$CVTF\_FROM\_INTERNAL\_TIME

## *resultant-time*

VMS usage: **floating\_point**  
type: **F-floating**  
access: **write only**  
mechanism: **by reference**

The external time that results from the conversion. The **resultant-time** argument is the address of an F-floating point value containing the result.

## *input-time*

VMS usage: **date\_time**  
type: **quadword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Delta time to be converted. The **input-time** argument is the address of an unsigned quadword containing the time.

---

## DESCRIPTION

LIB\$CVTF\_FROM\_INTERNAL\_TIME converts a delta internal VMS system time into an external F-floating point time. The **operation** argument specifies the conversion. LIB\$\_CVTF\_FROM\_INTERNAL\_TIME converts the value of **input-time** into one of the external formats listed in the **operation** argument description. LIB\$\_CVTF\_FROM\_INTERNAL\_TIME then places the result into **resultant-time**.

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Normal successful completion.
LIB\$_DELTIMREQ	Delta time required but absolute time supplied.
LIB\$_IVTIME	Invalid time.
LIB\$_WRONUMARG	Incorrect number of arguments.
LIB\$_INVOPER	Invalid operation.

# LIB\$CVT\_TO\_INTERNAL\_TIME

---

## LIB\$CVT\_TO\_INTERNAL\_TIME      Convert External Time to Internal Time

The Convert External Time to Internal Time routine converts an external time interval into a VMS internal format delta time.

---

**FORMAT**                    **LIB\$CVT\_TO\_INTERNAL\_TIME**    *operation ,input-time ,resultant-time*

---

**RETURNS**                    VMS usage: **cond\_value**  
                                  type:            **longword (unsigned)**  
                                  access:         **write only**  
                                  mechanism:     **by value**

---

**ARGUMENTS**                 **operation**  
                                  VMS usage: **function\_code**  
                                  type:            **longword (unsigned)**  
                                  access:         **read only**  
                                  mechanism:     **by reference**

The conversion to be performed. The **operation** argument is the address of an unsigned longword specifying the operation. Valid values for **operation** are the following:

---

<b>Operation</b>	<b>Interpretation</b>
LIB\$_DELTA_WEEKS	Whole weeks in delta time
LIB\$_DELTA_DAYS	Whole days in delta time
LIB\$_DELTA_HOURS	Whole hours in delta time
LIB\$_DELTA_MINUTES	Whole minutes in delta time
LIB\$_DELTA_SECONDS	Whole seconds in delta time

---

**input-time**  
VMS usage: **varying\_arg**  
type:            **longword (signed)**  
access:         **read only**  
mechanism:     **by reference**

Delta time to be converted. The **input-time** argument is the address of this input time. The value you supply for **input-time** must neither be negative nor greater than 10,000 days.

# LIB\$CVT\_TO\_INTERNAL\_TIME

## *resultant-time*

VMS usage: **date\_time**  
type: **quadword (unsigned)**  
access: **write only**  
mechanism: **by reference**

The VMS internal format delta time that results from the conversion. The **resultant-time** argument is the address of an unsigned quadword containing the result.

---

## DESCRIPTION

LIB\$CVT\_TO\_INTERNAL\_TIME converts an external time interval, such as three weeks, into a VMS internal format delta time. The **operation** argument specifies the conversion. LIB\$\_CVT\_TO\_INTERNAL\_TIME converts the value of **input-time** into one of the internal format delta times listed in the **operation** argument description. LIB\$\_CVT\_TO\_INTERNAL\_TIME then places the result into **resultant-time**.

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Normal successful completion.
LIB\$_IVTIME	Invalid time.
LIB\$_WRONUMARG	Incorrect number of arguments.
LIB\$_INVOPER	Invalid operation.

# LIB\$CVTF\_TO\_INTERNAL\_TIME

---

## LIB\$CVTF\_TO\_INTERNAL\_TIME      Convert External Time to Internal Time (F-Floating Point Value)

The Convert External Time to Internal Time (F-Floating Point Value) routine converts an external time interval into a VMS internal format F-floating delta time.

---

**FORMAT**                  LIB\$CVTF\_TO\_INTERNAL\_TIME    *operation*  
  *,input-time*  
  *,resultant-time*

---

**RETURNS**                  VMS usage: **cond\_value**  
                                  type:           **longword (unsigned)**  
                                  access:       **write only**  
                                  mechanism: **by value**

---

**ARGUMENTS**                *operation*  
                                  VMS usage: **function\_code**  
                                  type:           **longword (unsigned)**  
                                  access:       **read only**  
                                  mechanism: **by reference**

The conversion to be performed. The **operation** argument is the address of an unsigned longword specifying the operation. Valid values for **operation** are the following:

---

<b>Operation</b>	<b>Interpretation</b>
LIB\$_DELTA_WEEKS_F	Fractional weeks
LIB\$_DELTA_DAYS_F	Fractional days
LIB\$_DELTA_HOURS_F	Fractional hours
LIB\$_DELTA_MINUTES_F	Fractional minutes
LIB\$_DELTA_SECONDS_F	Fractional seconds

---

# LIB\$CVTF\_TO\_INTERNAL\_TIME

## *input-time*

VMS usage: **varying\_arg**  
type: **F-floating**  
access: **read only**  
mechanism: **by reference**

Delta time to be converted. The **input-time** argument is the address of this input time. The value you supply for **input-time** must neither be negative nor greater than 10,000 days.

## *resultant-time*

VMS usage: **date\_time**  
type: **quadword (unsigned)**  
access: **write only**  
mechanism: **by reference**

The VMS internal format delta time that results from the conversion. The **resultant-time** argument is the address of an unsigned quadword containing the result.

---

## DESCRIPTION

LIB\$CVTF\_TO\_INTERNAL\_TIME converts an external time interval, such as 3.5 weeks, into a VMS internal format F-floating delta time. The **operation** argument specifies the conversion. LIB\$\_CVTF\_TO\_INTERNAL\_TIME converts the value of **input-time** into one of the internal format delta times listed in the **operation** argument description. LIB\$\_CVTF\_TO\_INTERNAL\_TIME then places the result into **resultant-time**.

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Normal successful completion.
LIB\$_IVTIME	Invalid time.
LIB\$_WRONUMARG	Incorrect number of arguments.
LIB\$_INVOPER	Invalid operation.

# LIB\$CVT\_xTB

---

## LIB\$CVT\_xTB Convert Numeric Text to Binary

The Convert Numeric Text to Binary routines return a binary representation of the ASCII text string representation of a decimal, hexadecimal, or octal number.

---

<b>FORMAT</b>	<b>LIB\$CVT_DTB</b> <i>byte-count ,numeric-string ,result</i>
	<b>LIB\$CVT_HTB</b> <i>byte-count ,numeric-string ,result</i>
	<b>LIB\$CVT_OTB</b> <i>byte-count ,numeric-string ,result</i>

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b>
	type: <b>longword (unsigned)</b>
	access: <b>write only</b>
	mechanism: <b>by value</b>

---

<b>ARGUMENTS</b>	<b><i>byte-count</i></b>
	VMS usage: <b>longword_signed</b>
	type: <b>longword integer (signed)</b>
	access: <b>read only</b>
	mechanism: <b>by value</b>

Byte count of the input ASCII text string. The **byte-count** argument is a signed longword integer containing the byte count of the input string.

### ***numeric-string***

VMS usage:	<b>char_string</b>
type:	<b>character string</b>
access:	<b>read only</b>
mechanism:	<b>by reference</b>

ASCII text string representation of a decimal, hexadecimal, or octal number which LIB\$CVT\_xTB converts to binary representation. The **numeric-string** argument is the address of a character string containing this input string to be converted.

### ***result***

VMS usage:	<b>longword_signed</b>
type:	<b>longword integer (signed)</b>
access:	<b>write only</b>
mechanism:	<b>by reference</b>

Binary representation of the input string. The **result** argument is the address of a signed longword integer containing the converted string.

---

**DESCRIPTION**

LIB\$CVT\_DTB converts the ASCII text string representation of a decimal number into binary representation. LIB\$CVT\_HTB converts the ASCII text string representation of a hexadecimal number into binary representation. LIB\$CVT\_OTB converts the ASCII text string representation of an octal number into binary representation.

**Note:** LIB\$CVT\_DTB, LIB\$CVT\_HTB, and LIB\$CVT\_OTB are intended to be called primarily from BLISS and MACRO programs. Therefore, the routines expect input scalar arguments to be passed by value and strings by reference. Blanks are invalid characters.

---

**CONDITION  
VALUES  
RETURNED**

1  
0

Routine successfully completed.

Nonradix character in the input string or a sign in any position other than the first character. Blanks and tabs are invalid characters. An overflow from 32 bits (unsigned) will cause an error.



# LIB\$CVT\_VECTIM

---

## LIB\$CVT\_VECTIM Convert Seven-Word Vector to Internal Time

The Convert Seven-Word Vector to Internal Time routine converts a seven-word vector into a VMS internal format delta or absolute time.

---

**FORMAT**            **LIB\$CVT\_VECTIM** *input-time ,resultant-time*

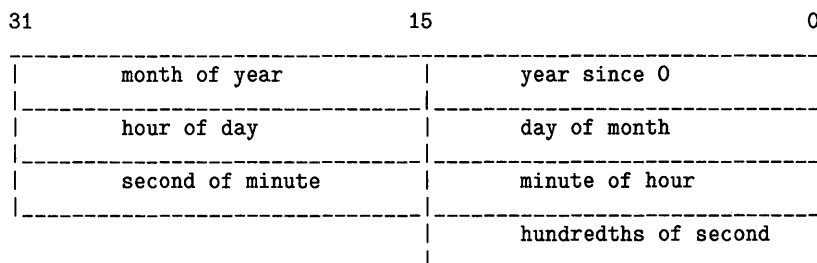
---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        ***input-time***  
                          VMS usage: **vector\_word\_unsigned**  
                          type:        **word (unsigned)**  
                          access:     **read only**  
                          mechanism: **by reference, array reference**

Time to be converted. The **input-time** argument is the address of a seven-word structure containing this time. This vector directly corresponds to a \$NUMTIM *timbuf* structure. The following diagram depicts the fields in this structure.



**Input-time** can represent an absolute or a delta time. In order for **input-time** to represent a delta time, the **year since 0** and **month of year** fields must equal zero. If those fields do not equal zero, an absolute time is returned.

***resultant-time***  
VMS usage: **date\_time**  
type:        **quadword (unsigned)**  
access:     **write only**  
mechanism: **by reference**

The VMS internal format delta or absolute time that results from the conversion. The **resultant-time** argument is the address of an unsigned quadword containing the result.

---

**DESCRIPTION** LIB\$CVT\_VECTIM converts a seven-word vector (in the format output by the system service SYS\$NUMTIM) into a VMS internal format delta or absolute time. LIB\$CVT\_VECTIM then places the result into **resultant-time**.

See the *VMS System Services Reference Manual* for more information about SYS\$NUMTIM.

---

<b>CONDITION VALUES RETURNED</b>	LIB\$_NORMAL	Normal successful completion.
	LIB\$_IVTIME	Invalid time.
	LIB\$_WRONUMARG	Incorrect number of arguments.

# LIB\$DATE\_TIME

---

## LIB\$DATE\_TIME Date and Time Returned as a String

The Date and Time Returned as a String routine returns the VMS system date and time in the semantics of a user-provided string.

---

**FORMAT**            **LIB\$DATE\_TIME** *date-time-string*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:           **longword (unsigned)**  
                          access:       **write only**  
                          mechanism:   **by value**

---

**ARGUMENT**            *date-time-string*  
                          VMS usage: **time\_name**  
                          type:           **character string**  
                          access:       **write only**  
                          mechanism:   **by descriptor**

Destination string into which LIB\$DATE\_TIME writes the system date and time. The **date-time-string** argument is the address of a descriptor pointing to the destination string.

This string is 23 characters long; its format is as follows:

dd-mmm-yyyy hh:mm:ss.hh

---

### CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Success, but destination string was truncated.
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_B_CLASS field.

---

## EXAMPLE

```
10 !+
   ! This BASIC program demonstrates the
   ! use of LIB$DATE_TIME.
   !-
   CALL LIB$DATE_TIME(DSTSTR$)
   PRINT DSTSTR$
99 END
```

This BASIC program uses LIB\$DATE\_TIME to display the current system date and time. The output generated by one run of this program was as follows:

```
26-JUL-1988 13:41:22.67
```

# LIB\$DAY

---

## LIB\$DAY Day Number Returned as a Longword Integer

The Day Number Returned as a Longword Integer routine returns the number of days since the system zero date of November 17, 1858, or the number of days from November 17, 1858, to a user-supplied date.

---

**FORMAT**            **LIB\$DAY** *number-of-days* [,*user-time*] [,*day-time*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:         **write only**  
                          mechanism:     **by value**

---

**ARGUMENTS**         ***number-of-days***  
                          VMS usage: **longword\_signed**  
                          type:            **longword integer (signed)**  
                          access:         **write only**  
                          mechanism:     **by reference**

Number of days since the system zero date. The **number-of-days** argument is the address of a signed longword integer containing the day number.

### ***user-time***

VMS usage: **date\_time**  
type:         **quadword (unsigned)**  
access:       **read only**  
mechanism:   **by reference**

User-supplied time, in 100-nanosecond units. The **user-time** argument is the address of a signed quadword integer containing the user time. A positive value indicates an absolute time, while a negative value indicates a delta time. This is an optional argument. If omitted, the default is the current system time. This quadword time value is obtained by calling the system service SYS\$BINTIM.

If time is passed as zero by value, the numeric value for the current day is returned. If time is passed as a zero by reference, the number returned represents the day of November 17th, 1858, rather than the current day.

### ***day-time***

VMS usage: **longword\_signed**  
type:         **longword integer (signed)**  
access:       **write only**  
mechanism:   **by reference**

Number of 10-millisecond units since midnight of the **user-time** argument. The **day-time** argument is the address of a signed longword integer into which LIB\$DAY writes this number of units.

---

**DESCRIPTION**

LIB\$DAY returns the number of days since the system zero date of November 17, 1858. Optionally, the caller can supply a time in system time format to be used instead of the current system time. In this case, LIB\$DAY returns the number of days from November 17, 1858, to the user-supplied date.

The number of 10-millisecond units since midnight is an optional return argument.

**Note: If the caller supplies a quadword time, it is not verified. If it is negative (bit 63 on), the number-of-days value returned is negative.**

The Run-Time Library provides the date/time utility routines for languages that do not have built-in time and date functions, and for particular applications that require the time or date in a different format from the one that the language supplies. In general, it is simpler to call the Run-Time Library routines for the system date and time than to call a system service.

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Routine successfully completed.

SS\$\_INTOVF

The optional argument **user-time** is present and represents a date past the year 8600.

---

**EXAMPLE**

```
PROGRAM DAY(INPUT, OUTPUT);
{+}
{ This is a VAX PASCAL example program showing
{ the use of LIB$DAY.
{-}
VAR
  DAYNUMBER : INTEGER;
routine LIB$DAY(VAR DAYNUM : INTEGER);
  EXTERN;
BEGIN
  LIB$DAY(DAYNUMBER);
  WRITELN('The daynumber is ', DAYNUMBER);
END.
```

A sample of the output generated by this program is shown below.

The daynumber is 46738

# LIB\$DAY\_OF\_WEEK

---

## LIB\$DAY\_OF\_WEEK Show Numeric Day of Week

The Show Numeric Day of Week routine returns the numeric day of the week for an input time value. If 0 is the input time value, the current day of the week is returned. The days are numbered 1 through 7, with Monday as day 1 and Sunday as day 7.

---

**FORMAT**            **LIB\$DAY\_OF\_WEEK** *user-time ,day-number*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        ***user-time***  
                          VMS usage: **date\_time**  
                          type:        **quadword (unsigned)**  
                          access:     **read only**  
                          mechanism: **by reference**

Time to be translated to a day of the week, or zero. The **user-time** argument is the address of an unsigned quadword containing the value of time. Time must be supplied as an absolute system time. To obtain this time value in proper quadword format call the system service SYS\$BINTIM.

If time is passed as zero by value, the numeric value for the current day is returned. If time is passed as a zero by reference, the number returned represents the day of November 17th, 1858, rather than the current day. If the **user-time** argument is omitted, it is equivalent to passing a zero by value.

***day-number***  
VMS usage: **longword\_unsigned**  
type:        **longword (unsigned)**  
access:     **write only**  
mechanism: **by reference**

Numeric day of week. The **day-number** argument is the address of a longword into which LIB\$DAY\_OF\_WEEK writes the integer value representing the day of the week.

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Routine successfully completed.

---

## EXAMPLE

```
PROGRAM DAYOFWEEK(INPUT, OUTPUT);
{+}
{ This is an example showing
{ the use of LIB$DAY_OF_WEEK.
{-}
VAR
  OUTDAT : INTEGER;
routine LIB$DAY_OF_WEEK(TIM : INTEGER; %REF OUTDA : INTEGER); EXTERN;
BEGIN
  LIB$DAY_OF_WEEK(%IMMED 0, OUTDAT);
  WRITELN(OUTDAT);
END.
```

This Pascal program shows the use of LIB\$DAY\_OF\_WEEK. This example was tested on a Monday, and the output generated was "1".



# LIB\$DECODE\_FAULT

---

## LIB\$DECODE\_FAULT Decode Instruction Stream During Fault

The Decode Instruction Stream During Fault routine is a tool for building condition handlers that process instruction fault exceptions. It is called from a condition handler.

---

<b>FORMAT</b>	<b>LIB\$DECODE_FAULT</b>	<i>signal-arguments</i> <i>,mechanism-arguments</i> <i>,user-procedure</i> <i>[,unspecified-user-argument]</i> <i>[,instruction-definitions]</i>
---------------	--------------------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>signal-arguments</i></b> VMS usage: <b>vector_longword_unsigned</b> type: <b>unspecified</b> access: <b>read only</b> mechanism: <b>by reference, array reference</b>
------------------	---

Signal arguments array that was passed from VMS to your condition handler. The **signal-arguments** argument is the address of the signal arguments array.

***mechanism-arguments***  
VMS usage: **vector\_longword\_unsigned**  
type: **unspecified**  
access: **read only**  
mechanism: **by reference, array reference**

Mechanism arguments array that was passed from VMS to your condition handler. The **mechanism-arguments** argument is the address of the mechanism arguments array.

***user-procedure***  
VMS usage: **procedure**  
type: **bound procedure value or procedure entry mask**  
access: **call after stack unwind**  
mechanism: **by descriptor, procedure descriptor**

User-supplied action routine that LIB\$DECODE\_FAULT calls to handle the exception. The **user-procedure** argument is the address of a descriptor pointing to your user action routine. **user-procedure** may be of type "bound procedure value" when called by languages with up-level addressing. If **user-procedure** is not of type "bound routine value," it is assumed to be the address of an entry mask.

# LIB\$DECODE\_FAULT

For further information on the user action routine, see "Call Format for a User Action Routine" in the Description section.

## ***unspecified-user-argument***

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

Additional information passed from your handler without interpretation to your user action routine. The **unspecified-user-argument** argument contains the value of this additional information. This is an optional argument; if omitted, zero is used.

## ***instruction-definitions***

VMS usage: **vector\_byte\_unsigned**  
type: **byte (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Array of bytes specifying instruction opcodes and operand definitions which are to replace or supplement the standard instruction definitions. The **instruction-definitions** argument is the address of this array.

If omitted, only the standard instruction definitions are used. If supplied, **instruction-definitions** is searched first, followed by the standard definitions.

Each instruction definition consists of a series of bytes, the first one or two of which is the instruction opcode. If the instruction is a 2-byte opcode, the escape byte, which must be hex FD, FE, or FF, is placed in the first of the two bytes. Following the opcode may be from 0 to 16 operand definition bytes. These bytes indicate the operand's access type and data type.

The end of each instruction definition is denoted by a byte containing the value LIB\$K\_DCFOPR\_END (zero). The list of instruction definitions is terminated by two bytes, each of which contains the value -1, (hexadecimal FF). For further information, see "Instruction Operand Definition Codes" in the Description section.

---

## **DESCRIPTION**

The Description section of the LIB\$DECODE\_FAULT routine is divided into five parts.

- Guidelines for Using LIB\$DECODE\_FAULT
- Exceptions Recognized by LIB\$DECODE\_FAULT
- Instruction Operand Definition Codes
- Call Format for a User Action Routine
- Call Format for a Signal Routine

# LIB\$DECODE\_FAULT

## Guidelines for Using LIB\$DECODE\_FAULT

LIB\$DECODE\_FAULT is a tool for building condition handlers that process instruction fault exceptions. Called from a condition handler, LIB\$DECODE\_FAULT performs the following actions.

- 1 Unwinds intermediate stack frames back to that of the exception
- 2 Decodes the instruction stream to determine the operation and its operands
- 3 Calls a user-supplied action routine and passes it a consistent and easy-to-access description of the instruction's context

Your user action routine performs whatever tasks are necessary to handle the fault and returns to LIB\$DECODE\_FAULT. LIB\$DECODE\_FAULT then restores the context as modified by your user action routine and continues execution.

Your condition handler must first decide whether or not it wishes to handle the exception. The signal arguments list contains the exception code and the address of the PC that is usually sufficient for this determination. Once LIB\$DECODE\_FAULT is called, if the exception is a fault LIB\$DECODE\_FAULT can analyze, control does not return to the condition handler. Therefore, your handler must not depend on regaining control by a routine return once it has called LIB\$DECODE\_FAULT. With your user action routine, LIB\$DECODE\_FAULT makes the original fault disappear.

**Note:** Your user action routine is capable of generating a new exception, including one that looks identical to the original exception. Your user action routine may also resignal, but if the decision to resignal is made inside the user action routine, all post-signal stack frames are lost.

Once your condition handler has decided that it wants to handle the exception, it calls LIB\$DECODE\_FAULT, passing as arguments the addresses of the signal and mechanism argument lists and a descriptor for your user action routine entry point. LIB\$DECODE\_FAULT will then perform the following actions:

- 1 Determine if the exception is a fault it understands. If not, it returns SS\$\_RESIGNAL.
- 2 Determine the context in which the exception occurred, including register and PSL contents, and save it.
- 3 Unwind all stack frames back to that frame in which the exception occurred.
- 4 Evaluate each operand's addressing mode, computing the resulting location for the operand. Immediate mode operands are expanded into their full form. If an invalid addressing mode is found, an SS\$\_RADRMOD exception is generated.
- 5 Unless the original exception was SS\$\_ACCVIO, test each operand for accessibility. If necessary, an access violation is signaled as if the instruction had tried to execute normally. See the paragraph following this list for more information.
- 6 Unless the original exception was SS\$\_ROPRAND, test each floating-point operand that is to be read for a reserved floating operand. If necessary, a reserved operand fault is signaled. See the paragraph following this list for more information.

# LIB\$DECODE\_FAULT

- 7 Determine the address of the next sequential instruction.
- 8 Call your user action routine with arguments as described below.
- 9 Upon return from your user action routine, reflect changes to the registers and PSL, and continue execution at the instruction address specified by your user action routine. Optionally, your user action routine may resignal the original exception.

Some instructions can generate more than one fault if evaluation of one operand causes a fault that occurs before a later operand (which would also cause a fault). An example of this is the possibility that a floating-point divide instruction might report a divide-by-zero fault upon seeing a zero divisor before noticing that the dividend was a reserved operand, or was inaccessible.

In these cases, operand-specific faults will be signaled immediately by LIB\$DECODE\_FAULT in the expectation that another condition handler (or the same one) can repair the situation. This may reorder the sequence of exceptions as seen by a program. If the operand exception is corrected, the original exception will reoccur, and the proper action will be taken.

If at all possible, you should attempt to determine if a resignal is necessary inside the condition handler that calls LIB\$DECODE\_FAULT, rather than inside your user action routine. The reason for this is that LIB\$DECODE\_FAULT removes all post-signal stack frames before calling your user action routine.

Your user action routine may fetch and store the operands, **registers** and **PSL** as is necessary for handling the exception. You should follow the VAX architecture rule of reading all input operands in left-to-right order, then writing all output operands in left-to-right order, to avoid inconsistent results with overlapping operands. This is especially necessary with register operands.

**PSL** may be modified in a manner consistent with the VAX architecture. If the T-bit in the PSL was set at the beginning of the instruction, LIB\$DECODE\_FAULT sets the TP bit. To initiate tracing, you must set only the T bit. To disable tracing, you must clear both the T and TP bits. See the *VAX Architecture Reference Manual* for more information.

If the first-part-done (FPD) bit in the PSL was set when the instruction faulted, LIB\$DECODE\_FAULT only advances the PC over the instruction; it does not reevaluate the operands and it sets **operand-count** to zero. It is assumed that if FPD is set, the operands are in known locations (typically the registers).

For the CASEB, CASEW, and CASEL instructions, only the **selector**, **base**, and **limit** operands are represented in **operand-count** and **read-operand-locations**. The element of registers that corresponds to the PC, described in the following text as R15, points to the first of the word-length displacements. Your user action routine must modify R15 to reflect the location of the next instruction to execute.

The standard instruction definitions used by LIB\$DECODE\_FAULT specify the XFC instruction (which causes an SS\$\_OPCCUS fault) as having zero operands. You may redefine XFC if needed using the **instruction-definitions** argument to LIB\$DECODE\_FAULT.

# LIB\$DECODE\_FAULT

If you do not want instruction execution to resume with the next sequential instruction, you must modify R15 appropriately. Your user action routine then returns to LIB\$DECODE\_FAULT which restores the registers and PSL, and resumes instruction execution. See also LIB\$\_RESTART below.

## Exceptions Recognized by LIB\$DECODE\_FAULT

LIB\$DECODE\_FAULT recognizes the following VAX faults:

- SS\$\_ACCVIO, access violation.
- SS\$\_BREAK, breakpoint fault.
- SS\$\_FLTDIV\_F, floating divide by zero.
- SS\$\_FLTOVF\_F, floating overflow.
- SS\$\_FLTUND\_F, floating underflow.
- SS\$\_OPCCUS, opcode reserved to customers and CSS.
- SS\$\_OPCDEC, opcode reserved to DIGITAL.
- SS\$\_ROPRAND, reserved operand.
- SS\$\_TBIT, T-bit pending trap. This is actually a fault caused by the TP bit being set at the beginning of instruction execution. It allows you to interpret all instructions by setting the PSL T-bit and allowing each instruction to trace-fault.

All other exceptions, including SS\$\_COMPAT and SS\$\_RADDRMOD, cause LIB\$DECODE\_FAULT to return immediately with the return status SS\$\_RESIGNAL.

SS\$\_COMPAT is generated by compatibility-mode instructions. LIB\$DECODE\_FAULT does not handle compatibility-mode instructions.

SS\$\_RADDRMOD is generated by a reserved addressing-mode fault. LIB\$DECODE\_FAULT assumes that all instructions follow VAX addressing-mode specifications.

## Instruction Operand Definition Codes

Each instruction operand has an access type (read, write, . . . ) and a data type (byte, word, . . . ) associated with it. The operand definition codes used in both the **instruction-definitions** argument passed to LIB\$DECODE\_FAULT and in the **operand-types** argument passed to the user action routine encode the access and data types in a byte. The fields and values for operand access and data types are described using the following symbols. These symbols are defined in DIGITAL-supplied symbol definition libraries as macro or module name \$LIBDCFDEF.

LIB\$\_DCFACC     The field of the operand description code that describes the operand access type (bits 0:2).

LIB\$\_DCFACC     The size of the access type field (3 bits).

# LIB\$DECODE\_FAULT

LIB\$M_DCFACC	<p>The mask for the access type field. This is a 3-bit field that can contain any binary value from 000 through 111. The integer value of these bit settings defines the operand access type code for the LIB\$M_DCFACC field. Currently, six codes are defined. These codes have symbolic names and are explained below. It is important to remember that LIB\$M_DCFACC is NOT a bit mask. The values 0 through 6 do not refer to bits 0 through 6. They represent the binary values 001 through 110 as contained in the 3-bit field.</p> <p>The operand access type codes defined for the LIB\$M_DCFACC field are:</p> <ul style="list-style-type: none"><li>LIB\$K_DCFACC_R = 1      Operand is read-only</li><li>LIB\$K_DCFACC_W = 2      Operand is write-only</li><li>LIB\$K_DCFACC_M = 3      Operand is to be modified</li><li>LIB\$K_DCFACC_A = 4      Operand is an address (must not be a register)</li><li>LIB\$K_DCFACC_V = 5      Operand is the base of a bit field (same as address except that it may be a register)</li><li>LIB\$K_DCFACC_B = 6      Operand is a branch address</li></ul>
LIB\$V_DCFTYP	<p>The field of the operand descriptor code that describes the operand data type (bits 3:7).</p>
LIB\$\$_DCFTYP	<p>The size of the operand data type field (5 bits).</p>
LIB\$M_DCFTYP	<p>The mask for the operand data type field. This is a 5-bit field (bits 3:7) that can contain any binary value from 00000 through 11111. The integer value of these bit settings defines the operand access type code for the LIB\$M_DCFACC field. Currently, nine codes are defined. These codes have symbolic names and are explained below. It is important to remember that LIB\$M_DCFTYP is NOT a bit mask. The values 0 through 9 do not refer to bits 0 through 9. They represent the binary values 00001 through 01001 as contained in the 5-bit field.</p> <p>The operand access type codes defined for the LIB\$V_DCFTYP field are:</p> <ul style="list-style-type: none"><li>LIB\$K_DCFTYP_B = 1      Operand is a byte</li><li>LIB\$K_DCFTYP_W = 2      Operand is a word</li><li>LIB\$K_DCFTYP_L = 3      Operand is a longword</li><li>LIB\$K_DCFTYP_Q = 4      Operand is a quadword</li><li>LIB\$K_DCFTYP_O = 5      Operand is an octaword</li><li>LIB\$K_DCFTYP_F = 6      Operand is an F_floating</li><li>LIB\$K_DCFTYP_D = 7      Operand is a D_floating</li><li>LIB\$K_DCFTYP_G = 8      Operand is a G_floating</li><li>LIB\$K_DCFTYP_H = 9      Operand is an H_floating</li></ul>

Symbols of the form LIB\$K\_DCFOPR\_xy, where x is the access type and y is the data type, are also defined. These combine the notions of access and data type. For example, LIB\$K\_DCFOPR\_MF has the following value:

51 (3+(6\*8))

# LIB\$DECODE\_FAULT

It denotes modify access of an F\_floating item. For the branch access type, only the types BB, BW, and BL are defined; otherwise, all combinations are available.

## Call Format for a User Action Routine

LIB\$DECODE\_FAULT calls the user action routine when it finds an exception to be handled. Your user action routine handles the exception in any manner that you specify and then returns to LIB\$DECODE\_FAULT.

```
action-routine  opcode ,instr-PC ,PSL ,registers ,operand-count
                  ,operand-types ,read-operand-locations
                  ,write-operand-locations ,signal-arguments
                  ,signal-procedure ,context
                  ,unspecified-user-argument ,original-registers
```

### opcode

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Opcode of the instruction that caused the fault. The **opcode** argument is the address of a longword that contains this opcode. LIB\$DECODE\_FAULT supplies this opcode when it calls the user action routine.

For 2-byte opcodes, the escape code (for example, hex FD) is in the low-order byte. You must use this argument to examine the opcode instead of reading the bytes pointed to by **instr-PC**. This is because if a debugger breakpoint has been set on the instruction, only **opcode** contains the original instruction.

### instr-PC

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Value of the PC for the instruction that caused the fault. The **instr-PC** argument is the address of a longword that contains the PC value.

Note the difference between this value and the contents of the **registers** array element that corresponds to the PC. R15 of the **registers** array element contains the address of the byte after the instruction that caused the fault.

### PSL

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

Processor status longword (PSL) at the time of the fault. The **PSL** argument is the address of a longword that contains this PSL. Your user action routine may modify this PSL within the restrictions of the VAX architecture.

### registers

VMS usage: **vector\_longword\_unsigned**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference, array reference**

# LIB\$DECODE\_FAULT

Contents of registers R0 through R15 (PC) at the time of the fault, but after operand addressing-mode processing. This includes any autoincrements and/or autodecrements. The **registers** argument is the address of this 16-longword array. Each longword of the registers array contains the contents of one register.

Your user action routine may modify these values. If it does, the new values will be reflected when instruction execution continues.

R15 denotes the 16th longword in the **registers** array, which corresponds to the PC. R15 contains the address of the next byte after the current instruction. Unless this value is modified by your user action routine, instruction execution will resume at that address. An exception is for the CASEB, CASEW, and CASEL instructions; R15 contains the address of the first displacement word. For these instructions, your user action routine must modify R15 to point to the next instruction to execute.

Upon instruction completion, registers R0-R15 are restored from this array. However, if **signal-procedure** is used to cause a fault or if instruction restart is specified by returning LIB\$\_RESTART, **original-registers** is used instead.

## **operand-count**

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Number of operands in the instruction currently being decoded. The **operand-count** is the address of a longword that contains this number.

## **operand-types**

VMS usage: **vector\_longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Array of longwords, one element for each operand, which contain the type codes for the associated operand. The **operand-types** argument is the address of this array.

The operand type codes are further defined under "Instruction Operand Definition Codes," which appeared above in this Description section.

## **read-operand-locations**

VMS usage: **vector\_longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Array of longwords, one element for each operand, which contains the addresses of the operands to be read. The **read-operand-locations** argument is the address of this array.

The address given in the array may not be the actual address of the operand if the operand is not a memory location. If the operand is a register, the address indicates a copy of the register value(s) at the time of operand evaluation. If the operand access type is ADDRESS or FIELD, and the operand is not a register, the address is the address of the item. If the operand access type is FIELD and the operand is a register, the address refers to the appropriate element in the **registers** array. If the operand access type is BRANCH, the



# LIB\$DECODE\_FAULT

address is the destination PC of the branch. For WRITE access operands, the address value is zero.

## **write-operand-locations**

VMS usage: **vector\_longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Array of longwords, one element for each operand, which contains the addresses of operands that are to be written. The **write-operand-locations** argument is the address of this array. If the operand access type is not MODIFY, WRITE, ADDRESS, or FIELD, the pointer value is zero.

## **signal-arguments**

VMS usage: **vector\_longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Signal arguments list of the original exception, as passed from VMS to your condition handler and then to LIB\$DECODE\_FAULT. The **signal-arguments** argument is the address of an array of longwords that contains these signal arguments.

## **signal-procedure**

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **call without stack unwinding**  
mechanism: **by reference**

Entry mask of a routine that your user action routine must call if it wishes to report an exception for the instruction that faulted. The **signal-procedure** argument is the address of this entry mask.

For further information, see "Call Format for a Signal Routine" in the Description section.

## **context**

VMS usage: **context**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

Context in which the exception occurs, including the register and PSL contents, to be used when calling the signal-procedure. The **context** argument contains the value of this context.

## **unspecified-user-argument**

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

Optional argument passed to LIB\$DECODE\_FAULT. If the argument was not specified, the value zero is substituted. The **unspecified-user-argument** argument contains the value of this optional argument.

# LIB\$DECODE\_FAULT

## original-registers

VMS usage: **vector\_longword\_unsigned**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference, array reference**

Array containing the values of registers R0 through R15 (PC) at the time of the fault, before operand processing. The **original-registers** argument is the address of this 16-longword array.

If the action routine specifies that the instruction should restart or that a fault should be generated, the registers are restored from **original-registers**. See also the description of **registers** above.

## Condition Values Returned from the User Action Routine

The user action routine can return the following condition values to LIB\$DECODE\_FAULT.

Condition Value	Description
SS\$_CONTINUE	If the user action routine returns a value of SS\$_CONTINUE, instruction execution will continue as specified by the current contents of the <b>registers</b> element for the PC.
SS\$_RESIGNAL	If it returns SS\$_RESIGNAL, the original exception is resignaled, with the only changes reflected being those specified by <b>registers</b> elements for R0 and R1 (which are stored in the mechanism arguments vector), PC, and PSL. All other registers are restored from original registers.
LIB\$_RESTART	If the user action routine returns LIB\$_RESTART, the current instruction is restarted with registers restored from <b>original-registers</b> and a PSL from <b>PSL</b> . This feature is useful for writing trace handlers.

## Call Format for a Signal Routine

Your action routine calls the signal routine using this format:

**signal-procedure** fault-flag ,context ,signal-arguments

### fault-flag

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Longword flag whose low-order bit determines whether or not the exception is to be signaled as a fault or as a trap. The **fault-flag** argument contains the address of this longword.

If the low-order bit of **fault-flag** is set to 1, the exception is signaled as a fault. If the low-order bit of **fault-flag** is set to 0, the exception is signaled as a trap; the current contents of the **registers** array are used. In either case, the current contents of **PSL** are used to set the exception PSL.

# LIB\$DECODE\_FAULT

## context

VMS usage: **context**  
type: **unspecified**  
access: **read only**  
mechanism: **by reference**

Context in which the new exception is to occur, as passed to your user action routine by LIB\$DECODE\_FAULT. The **context** argument is the address of this context value.

## signal-arguments

VMS usage: **arg\_list**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Signal arguments to be used. The **signal-arguments** argument is the address of an array of longwords that contains these signal arguments.

The first longword contains the number of following longwords; the remainder of the list contains signal names and arguments. Unlike the signal argument list passed to a condition handler, no PC or PSL is present.

Before the exception is signaled, the stack frames are unwound back to the original exception. You should be careful when causing a new signal that a loop of faults is not inadvertently generated. For example, the condition handler that called LIB\$DECODE\_FAULT will usually be called for the second signal. If the handler does not analyze the second signal as such, it may cycle through the identical path as for the first signal.

To resignal the current exception, have the user action routine return a value of SS\$\_RESIGNAL instead of calling the signal routine (unless you wish previously called condition handlers to be called again).

---

## CONDITION VALUES RETURNED

SS\$\_RESIGNAL

Resignal condition to next handler. The exception described by **signal-arguments** was not an instruction fault handled by LIB\$DECODE\_FAULT. If LIB\$DECODE\_FAULT can process the fault, it does not return to its caller.

---

## CONDITION VALUE SIGNALLED

LIB\$\_INVARG

Invalid argument to Run-Time Library. The instruction definition contained more than 16 operands or an operand definition contained an invalid data type or access code. This message is signaled after the stack frames have been unwound so that it appears to have been signaled from a routine that was called by the instruction that faulted.

---

**EXAMPLE**

```

C+
C Example condition handler and user-action routine using
C LIB$DECODE_FAULT. This example demonstrates the use of
C most of the features of LIB$DECODE_FAULT. Its purpose
C is to handle floating underflow and overflow faults,
C replacing the result of the instruction with the correctly
C signed smallest possible value for underflows, or greatest
C possible value for overflows.
C
C For simplicity, faults involving the POLYx instructions are
C not handled.
C
C***
C FIXUP_RESULT is the condition handler enabled by the program
C desiring the fixup of overflows and underflows.
C***
C-

      INTEGER*4 FUNCTION FIXUP_RESULT(SIGARGS, MECHARGS)

      IMPLICIT NONE
      INCLUDE '$SSDEF'           ! SS$ symbols
      INCLUDE '$LIBDCFDEF'       ! LIB$DECODE_FAULT symbols
      INTEGER*4 SIGARGS(1:*)     ! Signal arguments list
      INTEGER*4 MECHARGS(1:*)   ! Mechanism arguments list

C+
C This is a sample redefinition of MULH3 instruction.
C-

      BYTE OPTABLE(8) /'FD'X,'65'X,           ! MULH3 opcode
1          LIB$K_DCFOPR_RH,                 ! Read H_floating
2          LIB$K_DCFOPR_RH,                 ! Read H_floating
3          LIB$K_DCFOPR_WH,                 ! Write H_floating
4          LIB$K_DCFOPR_END,                ! End of operands
5          'FF'X,'FF'X/                    ! End of instructions

      INTEGER*4 LIB$DECODE_FAULT           ! External function
      EXTERNAL FIXUP_ACTION               ! Action routine to do the fixup

C+
C Determine if the exception is one we wish to handle.
C-

      IF ((SIGARGS(2) .EQ. SS$_FLTQVF_F) .OR.
1      (SIGARGS(2) .EQ. SS$_FLTUND_F)) THEN

C+
C We think we can handle the fault. Call
C LIB$DECODE_FAULT and pass it the signal arguments and
C the address of our action routine and opcode table.
C-

      FIXUP_RESULT = LIB$DECODE_FAULT (SIGARGS,
1      MECHARGS, %DESCR(FIXUP_ACTION),, OPTABLE)

      RETURN
      END IF

C+
C We can only get here if we couldn't handle the fault.
C Resignal the exception.
C-

```



# LIB\$DECODE\_FAULT

```

C+
C Table of operand sizes in 8-bit bytes, indexed by the
C datatype code contained in the OP_TYPES array. Only floating
C types matter.
C-

        BYTE OP_SIZES(9) /0,0,0,0,0,4,8,8,16/

        INTEGER*4 LIB$EXTV           ! External function
        INTEGER*4 RESULT_NEGATIVE    ! -1 if result negative,
                                        ! 0 if positive

        INTEGER*4 SIGN1,SIGN2,SIGN3  ! Signs of operands
        INTEGER*4 INST_BYTE          ! Current opcode byte
        INTEGER*4 INST_CLASS         ! Class of instruction
                                        ! from table

        INTEGER*4 OP_DTYPE           ! Datatype of operand
        INTEGER*4 OP_SIZE            ! Size of operand in
                                        ! 8-bit bytes

        INTEGER*4 RESULT_OP          ! Position of result
                                        ! in WRITE_OPS array

        LOGICAL*4 OVERFLOW           ! TRUE if SS$_FLTQVF_F
        LOGICAL*4 SMALLER            ! Function which
                                        ! compares operands

        PARAMETER ESCD = 'OFD'X      ! First byte of G,H instructions

        INTEGER*2 SMALL_F(2)         ! Smallest F_floating
        DATA SMALL_F /'0080'X,0/
        INTEGER*2 SMALL_D(4)         ! Smallest D_floating
        DATA SMALL_D /'0080'X,0,0,0/
        INTEGER*2 SMALL_G(4)         ! Smallest G_floating
        DATA SMALL_G /'0010'X,0,0,0/
        INTEGER*2 SMALL_H(8)         ! Smallest H_floating
        DATA SMALL_H /'0001'X,0,0,0,0,0,0,0/
        INTEGER*2 BIGGEST(8)         ! Biggest value (all datatypes)
        DATA BIGGEST /'7FFF'X,7*'FFFF'X/

        INTEGER*4 SIGNAL_ARRAY(2)    ! Array for signalling new
                                        ! exception

C+
C
C NOTE: Because the operands arrays contain the locations of
C the operands, rather than the operands themselves,
C we must call a routine using the %VAL function to
C "fool" the called routine into considering the
C contents of an operands array element as the address
C of an item. This would not be necessary in a
C language that understood the concept of pointer
C variables, such as PASCAL.
C
C If FPD is set in the PSL, signal SS$_ROPRAND (reserved operand). In
C reality this shouldn't happen since none of the instructions we
C handle can set FPD, but do it as an example.
C-

        IF (BTEST(PSL,PSL$V_FPD)) THEN
            SIGNAL_ARRAY(1) = 1        ! Count of signal arguments
            SIGNAL_ARRAY(2) = SS$_ROPRAND ! Error status value
            CALL SIGNAL_ROUT (
                1      1,                ! Fault flag - signal as fault
                2      SIGNAL_ARRAY,     ! Signal arguments array
                3      CONTEXT)          ! Context as passed to us
                                        ! Call will never return

        END IF

```

# LIB\$DECODE\_FAULT

```
C+
C Set OVERFLOW according to the exception type. We assume that
C the only alternatives are SS$_FLTOVF_F and SS$_FLTUND_F.
C-
      OVERFLOW = (SIGARGS(2) .EQ. SS$_FLTOVF_F)

C+
C Determine the datatype of the instruction by that of its
C second operand, since that is always the type of the
C destination.
C-
      OP_DTYPE = IBITS(OP_TYPES(2),LIB$_V_DCFTYP,LIB$_S_DCFTYP)

C+
C Get the size of the datatype in words.
C-
      OP_SIZE = OP_SIZES (OP_DTYPE)

C+
C Determine the class of instruction and dispatch to the
C appropriate routine.
C-
      INST_BYTE = IBITS(OPCODE,0,8)  ! Get first byte
      IF (INST_BYTE .EQ. ESCD) INST_BYTE = IBITS(OPCODE,8,8)
      INST_CLASS = INST_CLASS_TABLE(INST_BYTE)
      GO TO (1000,2000,3000,4000,5000,6000),INST_CLASS

C+
C If we get here, the instruction's entry in the
C INST_CLASS_TABLE is zero. This might happen if the instruction was
C a POLYx, or was some other unsupported instruction. Resignal the
C original exception.
C-
      FIXUP_ACTION = SS$_RESIGNAL      ! Resignal condition to next handler
      RETURN                          ! Return to LIB$DECODE_FAULT

C+
C 1000 - ADDF2, ADDF3, ADDD2, ADDD3, ADDG2, ADDG3, ADDH2, ADDH3
C
C Result's sign is the same as that of the first operand,
C unless this is an underflow, in which case the magnitudes of
C the values may change the sign.
C-
1000  RESULT_NEGATIVE = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      IF (.NOT. OVERFLOW) THEN
        IF (SMALLER(OP_SIZE,%VAL(READ_OPS(1)),
          1      %VAL(READ_OPS(2))))
          2  RESULT_NEGATIVE = .NOT. RESULT_NEGATIVE
        END IF
      GO TO 9000

C+
C 2000 - SUBF2, SUBF3, SUBD2, SUBD3, SUBG2, SUBG3, SUBH2, SUBH3
C
C Result's sign is the opposite of that of the first operand,
C unless this is an underflow, in which case the magnitudes of
C the values may change the sign.
C-
```

# LIB\$DECODE\_FAULT

```
2000  RESULT_NEGATIVE = .NOT. LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      IF (.NOT. OVERFLOW) THEN
          IF (SMALLER(OP_SIZE,%VAL(READ_OPS(1)),
                    1      %VAL(READ_OPS(2))))
              2  RESULT_NEGATIVE = .NOT. RESULT_NEGATIVE
          END IF
      GO TO 9000
```

C+

```
C 3000 - MULF2, MULF3, MULD2, MULD3, MULG2, MULG3, MULH2, MULH3,
C       DIVF2, DIVF3, DIVD2, DIVD3, DIVG2, DIVG3, DIVH2, DIVH3,
C
```

C If the signs of the first two operands are the same, then the  
C result's sign is positive, if they are not it is negative.

C-

```
3000  SIGN1 = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      SIGN2 = LIB$EXTV (15,1,%VAL(READ_OPS(2)))
      RESULT_NEGATIVE = SIGN1 .XOR. SIGN2
      GOTO 9000
```

C+

```
C 4000 - ACBF, ACBD, ACFG, ACBH
```

C

C The result's sign is the same as that of the second operand  
C (addend), unless this is underflow, in which case the  
C magnitudes of the addend and index may change the sign.  
C We must also determine if the branch is to be taken.

C-

```
4000  SIGN2 = LIB$EXTV (15,1,%VAL(READ_OPS(2)))
      RESULT_NEGATIVE = SIGN2
      IF (.NOT. OVERFLOW) THEN
          IF (SMALLER(OP_SIZE,%VAL(READ_OPS(2)),
                    1      %VAL(READ_OPS(3))))
              2  RESULT_NEGATIVE = .NOT. RESULT_NEGATIVE
          END IF
```

C+

C If this is overflow, then the branch is not taken, since the  
C result is always going to be greater or equal in magnitude  
C to the limit, and will be the correct sign. If underflow,  
C the branch is ALMOST always taken. The only case where the  
C branch might not be taken is when the result is exactly  
C equal to the limit. For this example, we are going to ignore  
C this exceptional case.

C-

```
      IF (.NOT. OVERFLOW)
          1  REGISTERS(15) = READ_OPS(4) ! Branch destination
      GO TO 9000
```

C+

```
C 5000 - CVTDF, CVTGF, CVTHF, CVTHD, CVTHG
```

C

C Result's sign is the same as that of the first operand.

C-

```
5000  RESULT_NEGATIVE = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      GO TO 9000
```

C+

```
C 6000 - EMOFDF, EMOFDD, EMOFDF, EMOFDD
```

C

C If the signs of the first and third operands are the same, then the  
C result's sign is positive, else it is negative.

C-



# LIB\$DECODE\_FAULT

```
6000  SIGN1 = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      SIGN2 = LIB$EXTV (15,1,%VAL(READ_OPS(3)))
      RESULT_NEGATIVE = SIGN1 .XOR. SIGN2
      GOTO 9000

C+
C All code paths merge here to store the result value. We also
C set the PSL appropriately. First, determine which operand is
C the result.
C-

9000  RESULT_OP = OP_COUNT
      IF (INST_CLASS .EQ. 4)
      1  RESULT_OP = RESULT_OP - 1      ! ACBx

C+
C      Select result based on datatype and exception type.
C-

      IF (OVERFLOW) THEN
      CALL LIB$MOVC3 (OP_SIZE,BIGGEST,%VAL(WRITE_OPS(RESULT_OP)))
      ELSE
      GO TO (9100,9200,9300,9400), OP_DTYPE-(LIB$K_DCFTYP_F-1)

C+
C      Should never get here. Resignal original exception.
C-

      FIXUP_ACTION = SS$_RESIGNAL
      RETURN

C+
C 9100 - F_floating result
C-

9100  CALL LIB$MOVC3 (OP_SIZE,SMALL_F,%VAL(WRITE_OPS(RESULT_OP)))
      GOTO 9500

C+
C 9200 - D_floating result
C-

9200  CALL LIB$MOVC3 (OP_SIZE,SMALL_D,%VAL(WRITE_OPS(RESULT_OP)))
      GOTO 9500

C+
C 9300 - G_floating result
C-

9300  CALL LIB$MOVC3 (OP_SIZE,SMALL_G,%VAL(WRITE_OPS(RESULT_OP)))
      GOTO 9500

C+
C 9400 - H_floating result
C-

9400  CALL LIB$MOVC3 (OP_SIZE,SMALL_H,%VAL(WRITE_OPS(RESULT_OP)))
      GOTO 9500

9500  END IF

C+
C Modify the PSL to reflect the stored result. If the result was
C negative, set the N bit. Clear the V (overflow) and Z (zero) bits.
C If the instruction was an ACBx, leave the C (carry) bit unchanged,
C otherwise clear it.
C-
```

# LIB\$DECODE\_FAULT

```
      IF (RESULT_NEGATIVE) THEN
        PSL = IBSET (PSL,PSL$V_N)      ! Set N bit
      ELSE
        PSL = IBCLR (PSL,PSL$V_N)      ! Clear N bit
      END IF
      PSL = IBCLR (PSL,PSL$V_V)        ! Clear V bit
      PSL = IBCLR (PSL,PSL$V_Z)        ! Clear Z bit
      IF (INST_CLASS .NE. 4)
        1 PSL = IBCLR (PSL,PSL$V_C)    ! Clear C bit if not ACBx

C+
C Set the sign of result.
C-

      IF (RESULT_NEGATIVE)
        1 CALL LIB$INSV (1,15,1,%VAL(WRITE_OPS(RESULT_OP)))

C+
C Fixup is complete. Return to LIB$DECODE_FAULT.
C-

      FIXUP_ACTION = SS$_CONTINUE
      RETURN
      END

C+
C Function which compares two floating values. It returns .TRUE. if
C the first argument is smaller in magnitude than the second.
C-

      LOGICAL*4 FUNCTION SMALLER(NBYTES,VAL1,VAL2)

      INTEGER*4 NBYTES          ! Number of bytes in values
      INTEGER*2 VAL1(*),VAL2(*)  ! Floating values to compare
      INTEGER*4 WORDA,WORDB

      SMALLER = .TRUE.          ! Initially return true

C+
C Zero extend to a longword for unsigned compares.
C Compare first word without sign bit.
C-

      WORDA = IBCLR(ZEXT(VAL1(1)),15)
      WORDB = IBCLR(ZEXT(VAL2(1)),15)
      IF (WORDA .LT. WORDB) RETURN

      DO I=2,NBYTES/2
        WORDA = ZEXT(VAL1(I))
        WORDB = ZEXT(VAL2(I))
        IF (WORDA .LT. WORDB) RETURN
      END DO

      SMALLER = .FALSE.         ! VAL1 not smaller than VAL2
      RETURN
      END
```

This FORTRAN example implements a simple recovery scheme for floating underflow and overflow faults, replacing the result of the instruction with the correctly signed smallest possible value for underflows or largest possible value for overflows.

---

## LIB\$DEC\_OVER Enable or Disable Decimal Overflow Detection

The Enable or Disable Decimal Overflow Detection routine enables or disables decimal overflow detection for the calling routine activation. The previous decimal overflow setting is returned.

---

**FORMAT**            **LIB\$DEC\_OVER** *new-setting*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                      type:       **longword integer (unsigned)**  
                      access:     **write only**  
                      mechanism: **by value**

The old decimal overflow enable setting (the previous contents of SF\$W\_PSW[PSW\$V\_DV] in the caller's frame).

---

**ARGUMENT**            *new-setting*  
                      VMS usage: **longword\_unsigned**  
                      type:       **longword (unsigned)**  
                      access:     **read only**  
                      mechanism: **by reference**

New decimal overflow enable setting. The **new-setting** argument is the address of an unsigned longword that contains the new decimal overflow enable setting. Bit 0 set to 1 means enable; bit 0 set to 0 means disable.

---

**DESCRIPTION**        The caller's stack frame will be modified by this routine.

A call to LIB\$DEC\_OVER affects only the current routine activation and does not affect any of its callers or any routines that it may call. However, the setting does remain in effect for any routines which are subsequently entered through a JSB entry point.

---

### EXAMPLE

```
DECOVF: ROUTINE OPTIONS (MAIN);
DECLARE LIB$DEC_OVER ENTRY (FIXED BINARY (7)) /* Address of byte for
                                                /* enable/disable
                                                /* setting */
        RETURNS (FIXED BINARY (31));          /* Old setting */
DECLARE DISABLE FIXED BINARY (7) INITIAL (0) STATIC READONLY;
DECLARE RESULT FIXED BINARY (31);
DECLARE (A,B) FIXED DECIMAL (4,2);
ON FIXEDOVERFLOW PUT SKIP LIST ('Overflow');
```

## LIB\$DEC\_OVER

```
RESULT = LIB$DEC_OVER (DISABLE);          /* Disable recognition of decimal
                                           /* overflow in this block          */

A = 99.99;
B = A + 2;
PUT SKIP LIST ('In MAIN');
  BEGIN;
  B = A + 2;
  PUT LIST ('In BEGIN block');
  CALL Q;
    Q: ROUTINE;
    B = A + 2;
    PUT LIST ('In Q');
    END Q;
  END /* Begin */;
END DECOVF;
```

This PL/I program shows how to use LIB\$DEC\_OVER to enable or disable the detection of decimal overflow. Note that in PL/I, disabling decimal overflow using this routine only causes the condition to be disabled in the current block; descendent blocks will enable the condition, unless this routine is called in each block.

# LIB\$DELETE\_FILE

---

## LIB\$DELETE\_FILE Delete One or More Files

The Delete One or More Files routine deletes one or more files. The specification of the file(s) to be deleted may include wildcards.

LIB\$DELETE\_FILE is similar in function to the DCL command DELETE.

---

<b>FORMAT</b>	<b>LIB\$DELETE_FILE</b> <i>filespec</i> [, <i>default-filespec</i> ] [, <i>related-filespec</i> ] [, <i>user-success-procedure</i> ] [, <i>user-error-procedure</i> ] [, <i>user-confirm-procedure</i> ] [, <i>user-specified-argument</i> ] [, <i>resultant-name</i> ] [, <i>file-scan-context</i> ]
---------------	---

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>filespec</i></b> VMS usage: <b>char_string</b> type: <b>character string</b> access: <b>read only</b> mechanism: <b>by descriptor</b>  String containing the VMS Record Management Services (RMS) file specification of the file(s) to be deleted. The <b>filespec</b> argument is the address of a descriptor pointing to the file specification. If the specification includes wildcards, each file that matches the specification is deleted. The string must not contain more than 255 characters. Any string class is supported.
------------------	---

### ***default-filespec***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Default file specification of the file(s) to be deleted. The **default-filespec** argument is the address of a descriptor pointing to the default file specification. This is an optional argument; if omitted, the default is the null string. Any string class is supported.

See the *VMS Record Management Services Manual* for information about default file specifications.

## ***related-filespec***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Related file specification of the file(s) to be deleted. The **related-filespec** argument is the address of a descriptor pointing to the related file specification. Any string class is supported. This is an optional argument; if omitted, the default is the null string.

Input file parsing is used. See the *VMS Record Management Services Manual* for information on related file specifications and input file parsing.

The related file specification is useful when you are processing lists of file specifications. Unspecified portions of the file specification are inherited from the last file processed.

## ***user-success-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied success routine that LIB\$DELETE\_FILE calls after it successfully deletes a file. The **user-success-procedure** argument is the address of the entry mask of the success routine.

The success routine can be used to display a log of the files that were deleted. For more information on the success routine, look under "Call Format for a Success Routine" in the Description section.

## ***user-error-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied error routine that LIB\$DELETE\_FILE calls when it detects an error. The **user-error-procedure** argument is the address of the entry mask of this routine.

The error routine returns a success/fail value which LIB\$DELETE\_FILE uses to determine if more files should be processed. For more information on the error routine, see "Call Format for an Error Routine" in the Description section.

## ***user-confirm-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied confirm routine that LIB\$DELETE\_FILE calls before each file is deleted. The **user-confirm-procedure** argument is the address of the entry mask of this routine. The value returned by the confirm routine determines whether or not the file will be deleted. The confirm routine can be used to select specific files for deletion based on criteria such as expiration date, size, and so on.

# LIB\$DELETE\_FILE

For more information about the confirm routine, see "Call Format for a Confirm Routine" in the Description section.

## ***user-specified-argument***

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

User-supplied argument that LIB\$DELETE\_FILE passes to the error, success and confirm routines each time they are called. Whatever mechanism is also used to pass **user-specified-argument** to LIB\$DELETE\_FILE is used to pass it to the routines. This is an optional argument; if omitted, zero is passed by value.

## ***resultant-name***

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

String into which LIB\$DELETE\_FILE writes the RMS resultant file specification of the last file processed. The **resultant-name** argument is the address of a descriptor pointing to the resultant name.

If present, **resultant-name** is used to store the file specification passed to the user-supplied routines, instead of a default class S, type T string. Therefore, this argument should be specified when the user-supplied routines are used and those routines require a descriptor type other than class S, type T. Any string class is supported.

If you are specifying one or more of the action routine arguments, be sure that the descriptor class used to pass **resultant-name** is the same as the descriptor class required by the action routine. For example, VAX Ada requires a class SB descriptor for string arguments to Ada routines, but will use a class A descriptor by default when calling external routines. Refer to your language manual to determine the proper descriptor class to use.

## ***file-scan-context***

VMS usage: **context**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

Context for deleting a list of file specifications. The **file-scan-context** argument is the address of a longword containing the context value.

You must initialize the file scan context to zero before the first of a series of calls to LIB\$DELETE\_FILE. LIB\$FILE\_SCAN uses this context to retain the file context for multiple input files. You must specify this context only when you are dealing with multiple input files, as the DCL command DELETE does. You may deallocate the context allocated by LIB\$FILE\_SCAN by calling LIB\$FILE\_SCAN\_END after all calls to LIB\$DELETE\_FILE have been completed.

**DESCRIPTION**

This Description section is divided into three parts.

- Call Format for a Success Routine
- Call Format for an Error Routine
- Call Format for a Confirm Routine

**Call Format for a Success Routine**

The success routine is called only if the **user-success-procedure** argument was specified in the LIB\$DELETE\_FILE argument list.

The calling format of a success routine is as follows:

```
user-success-procedure filespec [,user-specified-argument]
```

**filespec**

VMS usage: **char\_string**  
 type: **character string**  
 access: **read only**  
 mechanism: **by descriptor**

Record Management Services (RMS) resultant file specification of the file being deleted. The **filespec** argument is the address of a descriptor pointing to the file specification. If the **resultant-name** argument was specified, it is used to pass the string to the success routine. Otherwise, a class S, type T string is passed. Any string class is supported.

**user-specified-argument**

VMS usage: **user\_arg**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **unspecified**

Value of **user-specified-argument** passed by LIB\$DELETE\_FILE to the success routine. The same passing mechanism that was used to pass **user-specified-argument** to LIB\$DELETE\_FILE is used by LIB\$DELETE\_FILE to pass **user-specified-argument** to the success routine. This is an optional argument.

**Call Format for an Error Routine**

The error routine is called only if the **user-error-procedure** argument was specified in the LIB\$DELETE\_FILE argument list.

The calling format of an error routine is as follows:

```
user-error-procedure filespec ,rms-sts ,rms-stv ,error-source  

[,user-specified-argument]
```

**filespec**

VMS usage: **char\_string**  
 type: **character string**  
 access: **read only**  
 mechanism: **by descriptor**

String containing the RMS resultant file specification of the file being deleted. If **resultant-name** was specified, it is used to pass the string to the error routine. Otherwise, a class S, type T string is passed. Any string class is supported.



# LIB\$DELETE\_FILE

## **rms-sts**

VMS usage: **cond\_value**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Primary condition code (FAB\$L\_STS) which describes the error that occurred. The **rms-sts** argument is the address of an unsigned longword that contains the primary condition code.

## **rms-stv**

VMS usage: **cond\_value**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Secondary condition code (FAB\$L\_STV) which describes the error that occurred. The **rms-stv** argument is the address of an unsigned longword that contains the secondary condition code.

## **error-source**

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Integer code that indicates the point at which the error was found. The **error-source** argument is the address of a longword integer containing the code of the error source.

Possible values for the error code are as follows:

- 0 Error searching for file specification
- 1 Error deleting file

## **user-specified-argument**

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **unspecified**

Value passed to LIB\$DELETE\_FILE that is then passed to **user-error-procedure** using the same passing mechanism that was used to pass it to LIB\$DELETE\_FILE. This is an optional argument.

If the error routine returns a success status (bit 0 set), then LIB\$DELETE\_FILE will continue processing files. If a failure status (bit 0 clear) is returned, then processing will cease immediately and LIB\$DELETE\_FILE will return with the error status.

If the **user-error-procedure** argument is not specified, LIB\$DELETE\_FILE will return to its caller the most severe error status encountered while deleting the files. If the error routine is called for an error, the success status LIB\$\_ERRROUCAL is returned.

The error routine is not called for errors related to string copying.

## Call Format for a Confirm Routine

The confirm routine is called only if the **user-confirm-procedure** argument was specified in the call to LIB\$DELETE\_FILE.

The calling format of the confirm routine is as follows:

```
user-confirm-procedure filespec ,fab [,user-specified-argument]
```

### **filespec**

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

RMS resultant file specification of the file to be deleted. The **filespec** argument is the address of a descriptor pointing to the file specification.

If **resultant-name** was specified, it is used to pass the string to the confirm routine. Otherwise, a class S, type T string is passed. Any string class is supported.

### **fab**

VMS usage: **fab**  
type: **unspecified**  
access: **read only**  
mechanism: **by reference**

RMS file access block (FAB) that describes the file being deleted. Your program may perform an RMS \$OPEN on the FAB to obtain file attributes to determine whether the file should be deleted, but it must close the file with \$CLOSE before returning to LIB\$DELETE\_FILE.

### **user-specified-argument**

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **unspecified**

The value of the **user-specified-argument** argument that LIB\$DELETE\_FILE passes to the confirm routine using the same passing mechanism that was used to pass it to LIB\$DELETE\_FILE. This is an optional argument.

If confirm routine returns a success status (bit 0 set), the file is then deleted; otherwise, the file is not deleted.

# LIB\$DELETE\_FILE

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_ERRROUCAL	Success, but an error routine was called. A file error was encountered but the error routine was called to handle the condition.
LIB\$_INVFILSPE	Invalid file specification. <b>Filespec</b> or <b>default-filespec</b> is longer than 255 characters.
LIB\$_INVSTRDES	Invalid string descriptor. The descriptor for a string argument was not a valid string descriptor.
LIB\$_WRONUMARG	Wrong number of arguments. An incorrect number of arguments was passed to LIB\$DELETE_FILE.

Any condition value returned by LIB\$SCOPY\_xxx except those condition values specifying truncation errors.

Any condition value returned by RMS. If **user-error-procedure** is not specified, this is the most severe of the RMS errors encountered while deleting the files.

---

## EXAMPLE

```
PROGRAM DELETE_EXAMPLE(INPUT, OUTPUT);
{+}
{ Declare external function LIB$DELETE_FILE. Throughout this
{ example, the user-arg argument is not used.
{-}

FUNCTION LIB$DELETE_FILE(
    FILESPEC: VARYING [A] OF CHAR;
    DEFAULT_FILESPEC: VARYING [B] OF CHAR;
    REL_FILESPEC : VARYING [D] OF CHAR;
    %IMMED [UNBOUND] ROUTINE SUCCESS_ROUTINE
        (FILESPEC : VARYING [A] OF CHAR) := %IMMED 0;
    %IMMED [UNBOUND] FUNCTION ERROR_ROUTINE
        (FILESPEC : VARYING [A] OF CHAR; RMS_STS, RMS_STV : INTEGER)
        : BOOLEAN := %IMMED 0;
    %IMMED [UNBOUND] FUNCTION CONFIRM_ROUTINE
        (FILESPEC: VARYING [A] OF CHAR): BOOLEAN := %IMMED 0;
    VAR USER_ARG : [UNSAFE] INTEGER := %IMMED 0;
    VAR RESULT_NAME : VARYING [C] OF CHAR := %IMMED 0
) : INTEGER; EXTERN;

{+}
{ Declare a routine which will display the names of the files
{ as they are deleted.
{-}

ROUTINE LOG_ROUTINE(FILESPEC : VARYING [A] OF CHAR);
    BEGIN
        WRITELN('File ', FILESPEC, ' successfully deleted');
    END;

{+}
{ Declare a routine which will notify the user that an error
{ occurred.
{-}
```

## LIB\$DELETE\_FILE

```
FUNCTION ERR_ROUTINE(FILESPEC: VARYING [A] OF CHAR;
  RMS_STS, RMS_STV: INTEGER): BOOLEAN;
  BEGIN
    WRITELN('Delete of ', FILESPEC, ' failed ', HEX(RMS_STS));
    ERR_ROUTINE := TRUE;
  END;

{+}
{ Declare a routine which checks to see if the file should be
{ deleted.  If the filename contains the string 'XYZ', then it is
{ deleted.
{-}

FUNCTION CONFIRM_ROUTINE( FILESPEC: VARYING [A] OF CHAR): BOOLEAN;
  BEGIN
    IF INDEX(FILESPEC, 'XYZ') <> 0
    THEN
      CONFIRM_ROUTINE := TRUE
    ELSE
      CONFIRM_ROUTINE := FALSE;
  END;

{+}
{ The main program begins here.
{-}

VAR
  FILES_TO_DELETE, RESULTANT_NAME : VARYING [255] OF CHAR;
  RET_STATUS : INTEGER;

BEGIN
  WRITE ('Files to delete: ');
  READLN(FILES_TO_DELETE);
  RET_STATUS := LIB$DELETE_FILE(
    FILES_TO_DELETE, '*',', ', LOG_ROUTINE, ERR_ROUTINE,
    CONFIRM_ROUTINE,,RESULTANT_NAME);
  IF NOT ODD(RET_STATUS)
  THEN
    WRITELN('Delete failed.  The error was ', HEX(RET_STATUS));
END.
```

This Pascal program prompts the user for file specifications of files to be deleted. Of those, it deletes only files which contain the string 'XYZ' somewhere in their resultant file specification. The names of deleted files are displayed.

# LIB\$DELETE\_LOGICAL

---

## LIB\$DELETE\_LOGICAL Delete Logical Name

The Delete Logical Name routine requests the calling process's Command Language Interpreter (CLI) to delete a supervisor-mode process logical name. LIB\$DELETE\_LOGICAL provides the same function as the DCL command DEASSIGN.

---

**FORMAT**            **LIB\$DELETE\_LOGICAL** *logical-name* [, *table-name*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        ***logical-name***  
                          VMS usage: **logical\_name**  
                          type:        **character string**  
                          access:     **read only**  
                          mechanism: **by descriptor**

Logical name to be deleted. The **logical-name** argument is the address of a descriptor pointing to this logical name string. The maximum length of a logical name is 255 characters.

***table-name***  
VMS usage: **char\_string**  
type:        **character string**  
access:     **read only**  
mechanism: **by descriptor**

Name of the table from which the logical name is to be deleted. The **table-name** argument is the address of a descriptor pointing to this name string. This is an optional argument. If omitted, the LNM\$PROCESS table is used.

---

**DESCRIPTION**      LIB\$DELETE\_LOGICAL requests the calling process's Command Language Interpreter (CLI) to delete a supervisor-mode process logical name. If the optional **table-name** argument is defined, the logical name is deleted from that table. Otherwise, the logical name is deleted from the LNM\$PROCESS table.

Unlike the system service \$DELLOG and \$DELLNM, LIB\$DELETE\_LOGICAL does not require the caller to be executing in supervisor mode to delete a supervisor-mode logical name.

This routine is supported for use with the DCL and MCR Command Language Interpreters.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In that case, the error status LIB\$\_NOCLI is returned.

# LIB\$DELETE\_LOGICAL

See the *VMS DCL Dictionary* for a description of the DCL command DEASSIGN.

---

## CONDITION VALUES RETURNED

SS\$_ACCVIO	Access violation. The logical name could not be read.
SS\$_IVLOGNAM	Invalid logical name. The logical name contained illegal characters or more than 255 characters.
SS\$_IVLOGTAB	Invalid logical name table
SS\$_NOLOGNAM	No logical name match. The logical name was not defined as a supervisor-mode process logical name.
SS\$_NOPRIV	No privilege for attempted operation.
SS\$_NORMAL	Routine successfully completed.
SS\$_TOOMANYLNAM	Logical name translation exceeded allowed depth.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.
LIB\$_NOCLI	No CLI present to perform function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL Command Language Interpreter, please report the problem to DIGITAL by means of a Software Performance Report (SPR).

# LIB\$DELETE\_SYMBOL

---

## LIB\$DELETE\_SYMBOL Delete CLI Symbol

The Delete CLI Symbol routine requests the calling process's Command Language Interpreter (CLI) to delete an existing CLI symbol.

---

**FORMAT**            **LIB\$DELETE\_SYMBOL** *symbol* [, *table-type-indicator*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        *symbol*  
                          VMS usage: **char\_string**  
                          type:        **character string**  
                          access:     **read only**  
                          mechanism: **by descriptor**

Name of the symbol to be deleted by LIB\$DELETE\_SYMBOL. The **symbol** argument is the address of a descriptor pointing to this symbol string. The symbol name is converted to uppercase and trailing blanks are removed before use.

**Symbol** must begin with a letter, a dollar sign (\$), or an underscore (\_). The maximum length of **symbol** is 255 characters.

**table-type-indicator**  
VMS usage: **longword\_signed**  
type:        **longword integer (signed)**  
access:     **read only**  
mechanism: **by reference**

Indicator of the table which contains the symbol to be deleted. The **table-type-indicator** argument is the address of a signed longword integer that is this table indicator.

If omitted, the local symbol table is used. The following are possible values for the **table-type-indicator** argument.

---

Symbolic Name	Value	Table Used
LIB\$_CLI_LOCAL_SYM	1	Local symbol table
LIB\$_CLI_GLOBAL_SYM	2	Global symbol table

---

---

**DESCRIPTION**        LIB\$DELETE\_SYMBOL is supported for use with the DCL CLI. The error status LIB\$\_NOCLI will be returned if LIB\$DELETE\_SYMBOL is used with the MCR CLI or called from an image run directly as a subprocess or as a detached process.

# LIB\$DELETE\_SYMBOL

LIB\$K\_CLI\_LOCAL\_SYM and LIB\$K\_CLI\_GLOBAL\_SYM are defined in DIGITAL-supplied symbol libraries (macro or module name \$LIBCLIDEF) and as global symbols.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL.
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVARG	Invalid argument. The value of <b>table-type-indicator</b> was invalid.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.
LIB\$_INVSYMNAM	Invalid symbol name. The symbol name contained more than 255 characters or did not begin with a letter, a dollar sign, or an underscore.
LIB\$_NOCLI	No CLI present to perform the function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_NOSUCHSYM	No such symbol. The symbol was not defined.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL Command Language Interpreter, please report the problem to DIGITAL by means of a Software Performance Report (SPR).



# LIB\$DELETE\_VM\_ZONE

---

## LIB\$DELETE\_VM\_ZONE Delete Virtual Memory Zone

The Delete Virtual Memory Zone routine deletes a zone and returns all pages owned by the zone to the processwide page pool.

---

**FORMAT**            **LIB\$DELETE\_VM\_ZONE** *zone-id*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:          **write only**  
                          mechanism:       **by value**

---

**ARGUMENTS**        ***zone-id***  
                          VMS usage: **identifier**  
                          type:            **longword (unsigned)**  
                          access:          **read only**  
                          mechanism:       **by reference**

Zone identifier. The **zone-id** is the address of a longword that contains the identifier of a zone created by a previous call to LIB\$CREATE\_VM\_ZONE or LIB\$CREATE\_USER\_VM\_ZONE.

---

**DESCRIPTION**      LIB\$DELETE\_VM\_ZONE deletes a zone and returns all pages owned by the zone to the processwide page pool managed by LIB\$GET\_VM\_PAGE. The pages are then available for reallocation by later calls to LIB\$GET\_VM or LIB\$GET\_VM\_PAGE.

It takes less time to free memory in a single operation by calling LIB\$DELETE\_VM\_ZONE than to individually account for and free every block of memory that was allocated by calling LIB\$GET\_VM.

You must ensure that your program is no longer using any of the memory in the zone before you call LIB\$DELETE\_VM\_ZONE. Your program must not do any further operations on the zone after you call LIB\$DELETE\_VM\_ZONE.

If you specified deallocation filling when you created the zone, LIB\$DELETE\_VM\_ZONE will fill all of the allocated blocks that are freed.

If the zone you are deleting was created using the LIB\$CREATE\_USER\_VM\_ZONE routine, then you must have an appropriate action routine for the delete operation. That is, in your call to LIB\$CREATE\_USER\_VM\_ZONE, you must have specified a **user-delete-procedure**.

# LIB\$DELETE\_VM\_ZONE

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL  
LIB\$\_BADBLOADR

Normal successful completion.  
An invalid **zone-id** argument.

# LIB\$DIGIT\_SEP

---

## LIB\$DIGIT\_SEP Get Digit Separator Symbol

The Get Digit Separator Symbol routine returns the system's digit separator symbol.

---

**FORMAT**            **LIB\$DIGIT\_SEP** *digit-separator-string* [, *resultant-length*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        ***digit-separator-string***  
                          VMS usage: **char\_string**  
                          type:        **character string**  
                          access:     **write only**  
                          mechanism: **by descriptor**

Digit separator symbol returned by LIB\$DIGIT\_SEP. The **digit-separator-string** argument is the address of a descriptor pointing to the digit separator.

***resultant-length***  
VMS usage: **word\_unsigned**  
type:        **word (unsigned)**  
access:     **write only**  
mechanism: **by reference**

Number of characters written into **digit-separator-string**, not counting padding in the case of a fixed-length string. The **resultant-length** argument is the address of an unsigned word containing the length of the digit separator symbol. If the input string is truncated to the size specified in the **digit-separator-string** descriptor, **resultant-length** is set to this size. Therefore, **resultant-length** can always be used by the calling program to access a valid substring of **digit-separator-string**.

---

**DESCRIPTION**        LIB\$DIGIT\_SEP returns the symbol that is used to separate groups of three digits in the integer part of a number, for readability. A common digit separator is a comma (,) as in 3,006,854.

LIB\$DIGIT\_SEP attempts to translate the logical name SYS\$DIGIT\_SEP as a process, group, or system logical name. If the translation fails, LIB\$DIGIT\_SEP returns a comma (,), the United States digit separator. If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS\$DIGIT\_SEP as a system-wide logical name to provide a default for all users, and an individual user with a special need can define SYS\$DIGIT\_SEP as a process logical name to override the default symbol. For example, you may wish to use the French digit separator, the period (.).

BASIC implicitly uses LIB\$DIGIT\_SEP.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Successfully completed, but the digit separator string was truncated.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL.
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.

---

**EXAMPLE**

```

PROGRAM DIGIT_SEP(INPUT, OUTPUT);
{+}
{ This program uses LIB$DIGIT_SEP to return current
{ value of SYS$DIGIT_SEP.
{-}

routine LIB$DIGIT_SEP(%DESCR DIGIT_SEPSTR : VARYING [A]
  OF CHAR; %REF OUT_LEN : INTEGER); EXTERN;

VAR
  SEPARATOR : VARYING [256] OF CHAR;
  LENGTH : INTEGER;

BEGIN
  LIB$DIGIT_SEP(SEPARATOR, LENGTH);
  WRITELN('104',SEPARATOR,'567',SEPARATOR,'934');
END.

```

This Pascal example demonstrates how to use LIB\$DIGIT\_SEP. The output generated by this program is as follows:

104,567,934

# LIB\$DISABLE\_CTRL

---

## LIB\$DISABLE\_CTRL Disable CLI Interception of Control Characters

The Disable CLI Interception of Control Characters routine requests the calling process's Command Language Interpreter (CLI) to not intercept the selected control characters when they are typed during an interactive terminal session. LIB\$DISABLE\_CTRL provides the same function as the DCL command SET NOCONTROL.

---

**FORMAT**            **LIB\$DISABLE\_CTRL** *disable-mask* [*,old-mask*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:         **write only**  
                          mechanism:     **by value**

---

**ARGUMENTS**         ***disable-mask***  
                          VMS usage: **mask\_longword**  
                          type:            **longword (unsigned)**  
                          access:         **read only**  
                          mechanism:     **by reference**

Bit mask indicating which control characters are not to be intercepted. The **disable-mask** argument is the address of an unsigned longword containing this bit mask.

Each of the 32 bits corresponds to one of the 32 possible control characters. If a bit is set, the corresponding control character is no longer intercepted by the CLI. Currently, only bits 20 and 25, corresponding to CTRL/T and CTRL/Y, are recognized.

The following mask is defined in DIGITAL-supplied symbol libraries to specify the value of **disable-mask**.

---

Symbol	Hex Value	Function
LIB\$_CLI_CTRLT	%X'00100000'	Disables CTRL/T
LIB\$_CLI_CTRLY	%X'02000000'	Disables CTRL/Y

---

If a set bit does not correspond to a character which the CLI can intercept, LIB\$DISABLE\_CTRL returns an error.

***old-mask***  
VMS usage: **mask\_longword**  
type:            **longword (unsigned)**  
access:         **write only**  
mechanism:     **by reference**

# LIB\$DISABLE\_CTRL

Previous bit mask. The **old-mask** argument is the address of an unsigned longword into which LIB\$DISABLE\_CTRL writes the old bit mask. The old bit mask is of the same form as **disable-mask**.

---

## DESCRIPTION

The DCL and MCR CLIs can intercept the CTRL/Y control character. The DCL CLI can intercept the CTRL/T character. See the *VMS DCL Dictionary* for information on how the DCL CLI processes control characters, and see the *VAX-11 RSX Compatibility Mode Reference Manual* for information on how the MCR CLI processes control characters.

LIB\$DISABLE\_CTRL is supported for use with the DCL and MCR CLIs. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, LIB\$DISABLE\_CTRL returns the error status LIB\$\_NOCLI.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_INVARG	Invalid argument. A bit in <b>disable-mask</b> was set which did not correspond to a control character supported by the CLI.
LIB\$_NOCLI	No CLI present. Either the calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to DIGITAL by means of a Software Performance Report (SPR).

# LIB\$DO\_COMMAND

---

## LIB\$DO\_COMMAND Execute Command

The Execute Command routine stops program execution and directs the Command Language Interpreter to execute a command which you supply as the argument. If successful, LIB\$DO\_COMMAND does not return control to the calling program. Instead, LIB\$DO\_COMMAND begins execution of the specified command.

If you want control to return to the caller, use LIB\$SPAWN instead.

---

**FORMAT**            **LIB\$DO\_COMMAND** *command-string*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENT**            ***command-string***  
                          VMS usage: **char\_string**  
                          type:        **character string**  
                          access:     **read only**  
                          mechanism: **by descriptor**

Text of the command which LIB\$DO\_COMMAND executes. The **command-string** argument is the address of a descriptor pointing to the command text. The maximum length of the command is 255 characters.

---

**DESCRIPTION**        LIB\$DO\_COMMAND terminates your current image and then executes the contents of **command-string** as a command. The command is parsed using normal DCL rules.

LIB\$DO\_COMMAND is especially useful when you wish to execute a CLI command after your program has finished executing. For example, you could use the routine to execute a SUBMIT or PRINT command to handle a file that your program has created.

Because of the following restrictions on LIB\$DO\_COMMAND, you should be careful when you incorporate it in your program.

- During the call to LIB\$DO\_COMMAND, the current image exits and control cannot return to it.
- The text of the command is passed to the current Command Language Interpreter. Because you can define your own CLI in addition to DCL and MCR, you must make sure that the command will be handled by the intended CLI.
- If LIB\$DO\_COMMAND is called from an image run directly as a subprocess or detached process, it will not execute correctly, since no CLI is associated with a subprocess.

# LIB\$DO\_COMMAND

LIB\$DO\_COMMAND is supported for use with the DCL and MCR CLIs. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, the error status LIB\$\_NOCLI is returned. Note that the command can execute an indirect file using the at-sign (@) feature of DCL.

---

## CONDITION VALUES RETURNED

LIB\$_INVARG	Invalid argument. <b>Cmd-txt</b> was more than 255 characters.
LIB\$_NOCLI	No CLI present. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to DIGITAL by means of a Software Performance Report (SPR).

---

## EXAMPLE

```
PROGRAM DO_COMMAND(INPUT, OUTPUT);
{+}
{ This example uses LIB$DO_COMMAND to execute
{ any DCL command that is entered by the user
{ at the prompt.
{-}

ROUTINE LIB$DO_COMMAND(CMDTXT : VARYING [A] OF CHAR);
    EXTERN;

VAR
    COMMAND : VARYING [256] OF CHAR;

BEGIN
    WRITELN('ENTER THE COMMAND YOU WANT TO EXECUTE: ');
    READLN(COMMAND);
    LIB$DO_COMMAND(COMMAND);
END.
```

This Pascal program shows how to call LIB\$DO\_COMMAND. One example of the output of this program is as follows:

```
$ RUN DO_COMMAND
ENTER THE COMMAND YOU WANT TO EXECUTE: SHOW TIME
30-MAY-1988 14:07:28
```





---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Normal successful operation.
SS\$_INTOVF	Integer overflow. The quotient is replaced by bits 31:0 of the dividend, and the remainder is replaced by zero.
SS\$_INTDIV	Integer divide by zero. The quotient is replaced by bits 31:0 of the dividend, and the remainder is replaced by zero.

---

**EXAMPLE**

```

C+
C This FORTRAN program demonstrates how to use LIB$EDIV.
C-
      INTEGER DIVISOR,DIVIDEND(2),QUOTIENT,REMAINDER

C+
C Find the quotient and remainder of 4600387192 divided by 4096.
C Since 4600387192 is too large to store as a longword, use LIB$EDIV.
C-
      DIVISOR = 4096

C+
C The dividend must be represented as a quadword. To do this use a vector
C of length 2. The first element is the low order longword, and the second
C element is the high order longword.
C Now, 4600387192 = '00000000112345678'x. So,
C-
      DIVIDEND(1) = '12345678'X
      DIVIDEND(2) = '00000001'X

C+
C Compute the quotient and remainder of 4600387192 divided by 4096.
C-
      RETURN = LIB$EDIV(DIVISOR,DIVIDEND,QUOTIENT,REMAINDER)
      TYPE *, 'The longword integer quotient of 4600387192/4096 is:'
      TYPE *, '          ', QUOTIENT
      TYPE *, 'The longword integer remainder of 4600387192/4096 is:'
      TYPE *, '          ', REMAINDER
      END

```

This FORTRAN example demonstrates how to call LIB\$EDIV. The output generated by this program is as follows:

```

      The longword integer quotient of 4600387192/4096 is:
          1123141
      The longword integer remainder of 4600387192/4096 is:
          1656

```

# LIB\$EMODD

---

## LIB\$EMODD Extended Multiply and Integerize Routines for D-Floating Point Values

The Extended Multiply and Integerize routine (D-Floating Point Values) allows higher-level language users to perform accurate range reduction of D-floating arguments.

---

<b>FORMAT</b>	<b>LIB\$EMODD</b> <i>floating-point-multiplier</i> <i>,multiplier-extension</i> <i>,floating-point-multiplicand</i> <i>,integer-portion ,fractional-portion</i>
---------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>floating-point-multiplier</i></b> VMS usage: <b>floating_point</b> type: <b>D_floating</b> access: <b>read only</b> mechanism: <b>by reference</b>
------------------	--

The multiplier. The ***floating-point-multiplier*** argument is a D-floating number.

### ***multiplier-extension***

VMS usage: **byte\_unsigned**  
type: **byte (unsigned)**  
access: **read only**  
mechanism: **by reference**

The left-justified multiplier-extension bits. The ***multiplier-extension*** argument is an unsigned byte.

### ***floating-point-multiplicand***

VMS usage: **floating\_point**  
type: **D\_floating**  
access: **read only**  
mechanism: **by reference**

The multiplicand. The ***floating-point-multiplicand*** argument is a D-floating number.

***integer-portion***

VMS usage: **longword\_signed**  
 type: **longword integer (signed)**  
 access: **write only**  
 mechanism: **by reference**

The integer portion of the result. The **integer-portion** argument is the address of a signed longword integer containing the integer portion of the result.

***fractional-portion***

VMS usage: **floating\_point**  
 type: **D\_floating**  
 access: **write only**  
 mechanism: **by reference**

The fractional portion of the result. The **fractional-portion** argument is a D-floating number.

**DESCRIPTION**

The floating-point multiplier extension operand (second operand) is concatenated with the floating-point multiplier (first operand) to gain  $x$  additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a  $y$ -bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of  $y$  bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of  $x$  and  $y$  are listed below.

Routine	$x$	Bits	$y$
LIB\$EMODD	8	7:0	64

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
SS\$_INTOVF	Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS\$_INTOVF also.
SS\$_FLTUND	Floating underflow. The integer and fraction operands are replaced by zero.
SS\$_ROPRAND	Reserved operand. The integer and fraction operands are unaffected.



***integer-portion***

VMS usage: **longword\_signed**  
 type: **longword (signed)**  
 access: **write only**  
 mechanism: **by reference**

The integer portion of the result. The **integer-portion** argument is the address of a signed longword integer containing the integer portion of the result.

***fractional-portion***

VMS usage: **floating\_point**  
 type: **F\_floating**  
 access: **write only**  
 mechanism: **by reference**

The fractional portion of the result. The **fractional-portion** argument is the address of an F-floating number containing the fractional portion of the result.

**DESCRIPTION**

LIB\$EMODF allows higher-level language users to perform accurate range reduction of F-floating arguments.

The floating-point **multiplier-extension** operand (second operand) is concatenated with the **floating-point-multiplier** (first operand) to gain  $x$  additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a  $y$ -bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of  $y$  bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of  $x$  and  $y$  are listed below.

Routine	$x$	Bits	$y$
LIB\$EMODF	8	7:0	32

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Normal successful completion.
SS\$_INTOVF	Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS\$_INTOVF also.
SS\$_FLTUND	Floating underflow. The integer and fraction operands are replaced by zero.
SS\$_ROPRAND	Reserved operand. The integer and fraction operands are unaffected.



***integer-portion***

VMS usage: **longword\_signed**  
 type: **longword integer (signed)**  
 access: **write only**  
 mechanism: **by reference**

The integer portion of the result. The **integer-portion** argument is the address of a signed longword integer containing the integer portion of the result.

***fractional-portion***

VMS usage: **floating\_point**  
 type: **G\_floating**  
 access: **write only**  
 mechanism: **by reference**

The fractional portion of the result. The **fractional-portion** argument is a G-floating number.

**DESCRIPTION**

The floating-point multiplier extension operand (second operand) is concatenated with the floating-point multiplier (first operand) to gain  $x$  additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a  $y$ -bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of  $y$  bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of  $x$  and  $y$  are listed below.

Routine	x	Bits	y
LIB\$EMODG	11	15:5	64

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
SS\$_INTOVF	Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS\$_INTOVF also.
SS\$_FLTUND	Floating underflow. The integer and fraction operands are replaced by zero.
SS\$_ROPRAND	Reserved operand. The integer and fraction operands are unaffected.





The integer portion of the result. The **integer-portion** argument is the address of a signed longword integer containing the integer portion of the result.

***fractional-portion***

VMS usage: **floating\_point**  
 type: **H\_floating**  
 access: **write only**  
 mechanism: **by reference**

The fractional portion of the result. The **fractional-portion** argument is an H-floating number.

---

**DESCRIPTION**

The floating-point multiplier extension operand (second operand) is concatenated with the floating-point multiplier (first operand) to gain  $x$  additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a  $y$ -bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of  $y$  bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of  $x$  and  $y$  are listed below.

Routine	$x$	Bits	$y$
LIB\$EMODH	15	15:1	128

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
SS\$_INTOVF	Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS\$_INTOVF also.
SS\$_FLTUND	Floating underflow. The integer and fraction operands are replaced by zero.
SS\$_ROPRAND	Reserved operand. The integer and fraction operands are unaffected.



**product**

VMS usage: **quadword\_signed**  
 type: **quadword integer (signed)**  
 access: **write only**  
 mechanism: **by reference**

Product of the extended-precision multiplication. The **product** argument is the address of a signed quadword integer into which LIB\$EMUL writes the product.

**DESCRIPTION**

The multiplicand argument is multiplied by the multiplier argument giving a double-length result. The addend argument is sign-extended to double-length and added to the result. LIB\$EMUL then writes the result into the **product** argument.

For more information, see the *VAX Architecture Reference Manual*.

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Normal successful completion.

**EXAMPLE**

```

INTEGER MULT1,MULT2,ADDEND,PRODUCT(2)
C+
C Find the extended precision multiplication of 268435456 times 4096.
C That is, find the extended precision product of 2**28 times 2**12.
C Since 268435456 times 4096 is 2**40, a quadword value is needed for
C the calculation: use LIB$EMUL.
C-
MULT1 = 4096
MULT2 = 268435456
APPEND = 0
C+
C Compute 268435456*4096.
C Note that product will be stored as a quadword. This value will be stored
C in the 2 dimensional vector PRODUCT. The first element of PRODUCT will
C contain the low order bits, while the second element will contain the high
C order bits.
C-
RETURN = LIB$EMUL(MULT1,MULT2,APPEND,PRODUCT)
TYPE *,'PRODUCT(2) = ',PRODUCT(2),' and PRODUCT(1) = ',PRODUCT(1)
TYPE *,' '
TYPE *,'Note that 256 and 0 represent the hexadecimal value'
type *,'14H'1000000000'x,', which in turn, represents 2**40.'
END

```

This FORTRAN program demonstrates how to use LIB\$EMUL. The output generated by this program is as follows:

```
PRODUCT(2) =          256 and PRODUCT(1) =          0
```

Note that 256 and 0 represent the hexadecimal value '1000000000'x, which in turn represents 2<sup>40</sup>.

# LIB\$ENABLE\_CTRL

---

## LIB\$ENABLE\_CTRL Enable CLI Interception of Control Characters

The Enable CLI Interception of Control Characters routine requests the calling process's Command Language Interpreter (CLI) to resume interception of the selected control characters when they are typed during an interactive terminal session. LIB\$ENABLE\_CTRL provides the same function as the DCL command SET CONTROL.

---

**FORMAT**            **LIB\$ENABLE\_CTRL** *enable-mask* [,*old-mask*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        ***enable-mask***  
                          VMS usage: **mask\_longword**  
                          type:        **longword (unsigned)**  
                          access:     **read only**  
                          mechanism: **by reference**

Bit mask indicating for which control characters LIB\$ENABLE\_CTRL is to enable interception. The **enable-mask** argument is the address of an unsigned longword containing this bit mask. Each of the 32 bits corresponds to one of the 32 possible control characters. If a bit is set, the corresponding control character is intercepted by the CLI. Currently, only bits 20 and 25, corresponding to CTRL/T and CTRL/Y, are recognized.

The following mask is defined in DIGITAL-supplied symbol libraries to specify the value of **enable-mask**.

Symbol	Hex Value	Function
LIB\$_CLI_CTRLT	%X'00100000'	Enables CTRL/T
LIB\$_CLI_CTRLY	%X'02000000'	Enables CTRL/Y

If a set bit does not correspond to a character which the CLI can intercept, an error is returned.

***old-mask***  
VMS usage: **mask\_longword**  
type:        **longword (unsigned)**  
access:     **write only**  
mechanism: **by reference**

Previous bit mask. The **old-mask** argument is the address of an unsigned longword containing the old bit mask. The old bit mask is of the same form as **enable-mask**.

---

**DESCRIPTION**

LIB\$ENABLE\_CTRL provides the functions of the DCL SET CONTROL command. Normally, CTRL/Y interrupts the current command, command procedure, or image. After a call to LIB\$DISABLE\_CTRL, CTRL/Y is treated like CTRL/U followed by a carriage return. LIB\$ENABLE\_CTRL restores the normal operation of CTRL/Y or CTRL/T.

Both the DCL and MCR CLIs can intercept control characters. See the *VMS DCL Dictionary* for information on how the CLI processes control characters, and see the *VAX-11 RSX Compatibility Mode Reference Manual* for information on how the MCR CLI processes control characters.

LIB\$ENABLE\_CTRL is supported for use with the DCL or MCR CLIs.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, the error status LIB\$\_NOCLI is returned.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_INVARG	Invalid argument. A bit in <b>enable-mask</b> was set which did not correspond to a control character supported by the CLI.
LIB\$_NOCLI	No CLI present. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to DIGITAL by means of a Software Performance Report (SPR).

# LIB\$ESTABLISH

---

## LIB\$ESTABLISH Establish a Condition Handler

The Establish a Condition Handler routine moves the address of a condition handling routine (which can be a user-written or a library routine) to longword 0 of the stack frame of the caller of LIB\$ESTABLISH.

---

**FORMAT**            **LIB\$ESTABLISH** *new-handler*

---

**RETURNS**            VMS usage: **Routine**  
                          type:            **procedure entry mask**  
                          access:         **write only**  
                          mechanism:     **by reference**

Previous contents of SF\$A\_HANDLER (longword 0) of the caller's stack frame; zero if no handler existed.

---

**ARGUMENT**            *new-handler*  
                          VMS usage: **procedure**  
                          type:            **procedure entry mask**  
                          access:         **read only**  
                          mechanism:     **by value**

Routine to be set up as the condition handler. The **new-handler** argument is the address of the procedure entry mask to this routine.

---

**DESCRIPTION**        LIB\$ESTABLISH moves the address of a condition-handling routine to longword 0 of the stack frame of the caller of LIB\$ESTABLISH. This condition-handling routine then becomes the caller's condition handler. LIB\$ESTABLISH returns the previous contents of longword 0. This can either be the address of the caller's previous condition handler or zero if no handler existed.

The new condition handler remains in effect for your routine until you call LIB\$REVERT or until control returns to the caller of the routine that called LIB\$ESTABLISH. Once this happens, you must call LIB\$ESTABLISH again if the same (or a new) condition handler is to be associated with the routine that called LIB\$ESTABLISH.

LIB\$ESTABLISH modifies the caller's stack frame.

LIB\$ESTABLISH is provided primarily for use with languages without built-in error handling facilities. Do not use LIB\$ESTABLISH with languages that provide error handling, such as BASIC, COBOL, Pascal, and PL/I. Use of this routine with these languages may adversely affect the behavior of your program. The language-support library for these languages depends on predefined language-specific handlers. Also, the handler address is used to identify the stack frames of routines written in these languages. See the documentation for the language you are using for more information about how that language handles errors.

# LIB\$ESTABLISH

In MACRO, you merely use the following instruction instead of calling LIB\$ESTABLISH:

```
MOVAB HANDLER, (FP)      ; set handler address
                        ; in current stack frame
```

---

**CONDITION**      *None.*  
**VALUES**  
**RETURNED**

---

## EXAMPLE

```
C+
C This FORTRAN program demonstrates the
C use of LIB$ESTABLISH.
C
C This is the main program.
C-

      EXTERNAL LOG_HANDL
      CHARACTER TIMBUF
      OPEN (UNIT=99, FILE = 'ERRLOG', STATUS = 'NEW')
      CALL LIB$ESTABLISH (LOG_HANDL)

      CALL SYS$BINTIM (TIMBUF, TIMADR)

C+
C The rest of the main program would go here.
C-

      END

      INTEGER*4 FUNCTION LOG_HANDL (SIGARGS, MECHARGS)
      INTEGER*4 SIGARGS (*), MECHARGS (5)

C+
C This is the handler to journal any signaled error messages.
C-

      INCLUDE '($SDEF)'
      EXTERNAL PUT_LINE
      LOG_HANDL = SS$_RESIGNAL
      CALL SYS$PUTMSG (SIGARGS, PUT_LINE, )
      RETURN
      END

C+
C This is the action subroutine.
C-

      LOGICAL*4 FUNCTION PUT_LINE (LINE)
      CHARACTER*(*)LINE
      PUT_LINE = .FALSE.
100   WRITE (99,200)LINE
200   FORMAT (A)
      RETURN
      END
```

In this FORTRAN example, the function `log_handl` is the condition handler for the program, and thus receives control when an error occurs.



# LIB\$EXTV

---

## LIB\$EXTV Extract a Field and Sign-Extend

The Extract a Field and Sign-Extend routine returns a sign-extended longword field that has been extracted from the specified variable bit field. LIB\$EXTV makes the VAX EXTV instruction available as a callable routine.

---

**FORMAT**            **LIB\$EXTV** *position ,size ,base-address*

---

**RETURNS**            VMS usage: **longword\_signed**  
                          type:            **longword integer (signed)**  
                          access:        **write only**  
                          mechanism:    **by value**

Field extracted by LIB\$EXTV, sign-extended to a longword.

---

### ARGUMENTS

***position***  
VMS usage: **longword\_signed**  
type:        **longword integer (signed)**  
access:      **read only**  
mechanism: **by reference**

Position (relative to the base address) of the first bit in the field that LIB\$EXTV extracts. The **position** argument is the address of a signed longword integer containing the position.

***size***  
VMS usage: **byte\_unsigned**  
type:        **byte (unsigned)**  
access:      **read only**  
mechanism: **by reference**

Size of the bit field LIB\$EXTV extracts. The **size** argument is the address of an unsigned byte containing the size. The maximum size is 32 bits.

***base-address***  
VMS usage: **longword\_unsigned**  
type:        **longword (unsigned)**  
access:      **read only**  
mechanism: **by value**

Base address of the bit field LIB\$EXTV extracts from the specified variable bit field. The **base-address** argument is an unsigned longword containing this base address.

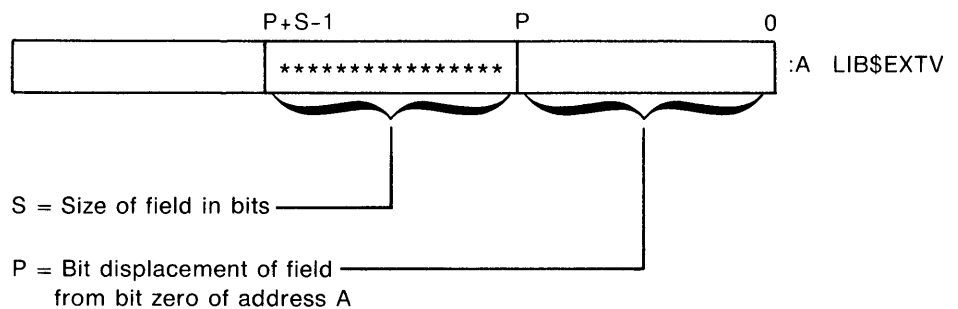
**DESCRIPTION**

The variable-length bit field is a VAX data type used to store small integers packed together in a larger data structure. It is often used to store single flag bits.

Three scalar attributes define a variable bit field.

- The base address is the address of a byte in memory that serves as a reference point for locating the bit field.
- The bit position is a signed longword containing the displacement of the least significant bit of the field with respect to the bit zero of the base address.
- The size is a byte integer indicating the size of the bit field in bits (in the range  $0 \leq \text{size} \leq 32$ ). That is, a bit field can be no more than one longword in length.

A variable-length bit field has the following format. The shaded area indicates the field.



ZK-1940-84

Bit fields are zero-origin, which means that the routine regards the first bit in the field as being the zero position. For more detailed information on VAX bit number and data formats, see the *VAX Architecture Reference Manual*.

The Run-Time Library routines for performing operations on variable-length bit fields give higher-level languages direct access to the bit field instructions in the VAX instruction set.

**CONDITION  
VALUE  
SIGNALLED**

SS\$\_ROPRAND

A reserved operand fault occurs if a size greater than 32 is specified.

# LIB\$EXTV

---

## EXAMPLE

```
SIGN_EXTEND: ROUTINE OPTIONS (MAIN);
DECLARE LIB$EXTV ENTRY
    (FIXED BINARY (31),      /* Address of longword containing
                           /* beginning bit position      */
    FIXED BINARY (7),      /* Address of byte containing size
                           /* of field                      */
    FIXED BINARY (31))     /* Address of field              */
    RETURNS (FIXED BINARY (31)); /* Return value                 */

DECLARE (VALUE, SMALL_INT) FIXED BINARY (31);
ON ENDFILE (SYSIN) STOP;

DO WHILE ('1'B);          /* Loop continuously, until end of file */
    PUT SKIP(2);
    GET LIST (VALUE) OPTIONS (PROMPT ('Value: '));
    SMALL_INT = LIB$EXTV ( 0, 4, VALUE); /* Extract and sign-extend
                                         /* first 4 bits                */
    PUT SKIP LIST (VALUE, SMALL_INT);
END;

END SIGN_EXTEND;
```

This PL/I program extracts a field and returns it sign-extended into a longword.

---

## LIB\$EXTZV Extract a Zero-Extended Field

The Extract a Zero-Extended Field routine returns a longword zero-extended field that has been extracted from the specified variable bit field. LIB\$EXTZV makes the VAX EXTZV instruction available as a callable routine.

---

**FORMAT**            **LIB\$EXTZV** *position ,size ,base-address*

---

**RETURNS**            VMS usage: **longword\_signed**  
                           type:        **longword integer (signed)**  
                           access:     **write only**  
                           mechanism: **by value**

Field extracted by LIB\$EXTZV, zero-extended to a longword.

---

### ARGUMENTS

***position***  
 VMS usage: **longword\_signed**  
 type:        **longword (signed)**  
 access:     **read only**  
 mechanism: **by reference**

Position (relative to the base address) of the first bit in the field LIB\$EXTZV extracts. The **position** argument is the address of a signed longword integer containing the position.

***size***  
 VMS usage: **byte\_unsigned**  
 type:        **byte (unsigned)**  
 access:     **read only**  
 mechanism: **by reference**

Size of the bit field LIB\$EXTZV extracts. The **size** argument is the address of an unsigned byte containing the size. The maximum size is 32 bits.

***base-address***  
 VMS usage: **longword\_unsigned**  
 type:        **longword (unsigned)**  
 access:     **read only**  
 mechanism: **by value**

Base address of the bit field LIB\$EXTZV extracts. The **base-address** argument is an unsigned longword containing this base address.

---

### DESCRIPTION

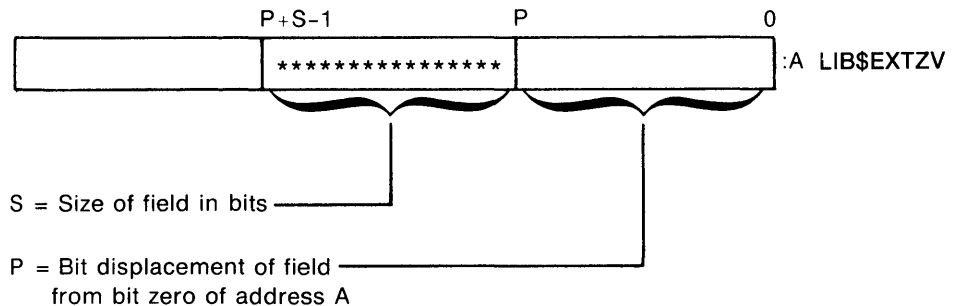
The variable-length bit field is a VAX data type used to store small integers packed together in a larger data structure. It is often used to store single flag bits.

# LIB\$EXTZV

Three scalar attributes define a variable bit field.

- The base address is the address of the byte in memory that serves as a reference point for locating the bit field.
- The bit position is a signed longword containing the displacement of the least significant bit of the field with respect to the bit zero of the base address.
- The size is a byte integer indicating the size of the bit field in bits (in the range  $0 \leq \text{size} \leq 32$ ). That is, a bit field can be no more than one longword in length.

A variable-length bit field has the following format. The shaded area indicates the field.



ZK-1941-84

Bit fields are zero-origin fields, which means that the routine regards the first bit in the field as being the zero position. For more detailed information on VAX bit numbering and data formats, see the *VAX Architecture Reference Manual*.

The Run-Time Library routines for performing operations on variable-length bit fields give higher-level languages direct access to the bit field instructions in the VAX instruction set.

---

## CONDITION VALUE SIGNALLED

SS\$\_ROPRAND

A reserved operand fault occurs if a size greater than 32 is specified.

---

## LIB\$FFx Find First Clear or Set Bit

The Find First Clear or Set Bit routines search the field specified by the start position, size, and base for the first clear or set bit. LIB\$FFC and LIB\$FFS make the VAX FFC and FFS instructions available as callable routines.

---

<b>FORMAT</b>	<b>LIB\$FFC</b> <i>position ,size ,base ,find-position</i>
	<b>LIB\$FFS</b> <i>position ,size ,base ,find-position</i>

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b>
	type: <b>longword (unsigned)</b>
	access: <b>write only</b>
	mechanism: <b>by value</b>

---

<b>ARGUMENTS</b>	<b><i>position</i></b>
	VMS usage: <b>longword_signed</b>
	type: <b>longword integer (signed)</b>
	access: <b>read only</b>
	mechanism: <b>by reference</b>

Starting position, relative to the base address, of the bit field to be searched by LIB\$FFx. The **position** argument is the address of a signed longword integer containing the starting position.

### ***size***

VMS usage:	<b>byte_unsigned</b>
type:	<b>byte (unsigned)</b>
access:	<b>read only</b>
mechanism:	<b>by reference</b>

Number of bits to be searched by LIB\$FFx. The **size** argument is the address of an unsigned byte containing the size of the bit field to be searched. The maximum size is 32 bits.

### ***base***

VMS usage:	<b>address</b>
type:	<b>longword (unsigned)</b>
access:	<b>read only</b>
mechanism:	<b>by reference</b>

Base address of the bit field which LIB\$FFx searches. The **base** argument is the address of an unsigned longword containing the base address.

# LIB\$FFx

## *find-position*

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **write only**  
mechanism: **by reference**

Bit position of the first bit in the specified state (clear or set), relative to the base address. The **find-position** argument is the address of a signed longword integer into which LIB\$FFC writes the position of the first clear bit and into which LIB\$FFS writes the position of the first set bit.

---

## DESCRIPTION

LIB\$FFC searches the field specified by the start position, size, and base for the first clear bit. LIB\$FFS searches the field for the first set bit.

If a bit in the specified state is found, LIB\$FFx writes the position (relative to the base) of that bit into **find-position** and returns a success status. If no bits are in the specified state or if **size** is zero, LIB\$FFx returns LIB\$\_NOTFOU and sets **find-position** to the starting position plus the size.

LIB\$FFx regards the first bit in the field as being the zero position. For more detailed information on VAX bit numbering and data formats, see the *VAX Architecture Reference Manual*.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. A bit in the specified state was found.
LIB\$_NOTFOU	A bit in the specified state was not found.

---

## CONDITION VALUE SIGNALLED

SS\$_ROPRAND	Reserved operand fault. A size greater than 32 was specified.
--------------	---

---

## LIB\$FID\_TO\_NAME Convert Device and File ID to File Specification

The Convert Device and File ID to File Specification routine converts a disk device name and file identifier to a file specification.

---

**FORMAT**            **LIB\$FID\_TO\_NAME**    *device-name ,file-id ,filespec  
[ ,filespec-length] [ ,directory-id]  
[ ,acp-status]*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:           **longword (unsigned)**  
                          access:         **write only**  
                          mechanism:     **by value**

---

### ARGUMENTS

#### ***device-name***

VMS usage: **char\_string**  
type:         **character string**  
access:       **read only**  
mechanism:   **by descriptor**

Device name to be converted. The **device-name** argument is the address of a descriptor pointing to the device name. **Device-name** must reference a disk device, and must contain 64 characters or less. LIB\$FID\_TO\_NAME obtains **device-name** from the NAM\$\_DVI field of a RMS name block.

#### ***file-id***

VMS usage: **vector\_word\_unsigned**  
type:         **word (unsigned)**  
access:       **read only**  
mechanism:   **by reference**

Specifies the file identifier. The **file-id** argument is the address of an array of three words containing the file identification. LIB\$FID\_TO\_NAME obtains **file-id** from the NAM\$\_FID field of a RMS name block. The \$FIDDEF macro defines the structure of **file-id**.

#### ***filespec***

VMS usage: **char\_string**  
type:         **character string**  
access:       **write only**  
mechanism:   **by descriptor**

Receives the file specification. The **filespec** argument is the address of a descriptor pointing to the file specification string.



# LIB\$FID\_TO\_NAME

## *filespec-length*

VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **write only**  
mechanism: **by reference**

Receives the number of characters written into **filespec**, excluding padding in the case of a fixed-length string. The optional **filespec-length** argument is the address of an unsigned word containing the number of characters.

If the output string is truncated to the number of characters specified in **filespec**, then **filespec-length** is set to that truncated size. Therefore, you can always use **filespec-length** to access a valid substring of **filespec**.

## *directory-id*

VMS usage: **vector\_word\_unsigned**  
type: **word (unsigned)**  
access: **read only**  
mechanism: **by reference, array reference**

Specifies a directory file identifier. The **directory-id** argument is the address of an array of three words containing the directory file identifier. LIB\$FID\_TO\_NAME obtains this array from the NAM\$W\_DID field of a RMS name block. The \$FIDDEF macro defines the structure of **directory-id**.

## *acp-status*

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by reference**

The status resulting from traversing the backward links. The optional **acp-status** argument is the address of an unsigned longword containing the status.

---

## DESCRIPTION

LIB\$FID\_TO\_NAME converts a disk device name and file identifier to a file specification by requesting the ACP file specification attribute. If you use the LIB\$FID\_TO\_NAME routine on a structure level 1 disk, specify the **directory-id** argument to ensure proper operation of the routine.

LIB\$FID\_TO\_NAME stores the output arguments (**filespec**, **filespec-length**, and **acp-status**) only if the routine successfully finishes.

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Normal successful completion.
LIB\$_STRTRU	Output string truncated (qualified success).
LIB\$_INVARG	Required argument omitted, or <b>device-name</b> is longer than 64 characters.
LIB\$_INVFILSPE	<b>Device-name</b> does not reference a disk.

Any condition value returned by LIB\$ANALYZE\_SDESC, SYS\$ASSIGN, SYS\$QIO, or SYS\$DASSGN.



# LIB\$FILE\_SCAN

## **context**

VMS usage: **context**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

Default file context used in processing file specifications for multiple input files. The **context** argument is the address of a longword, which must be initialized to zero by your program before the first call to LIB\$FILE\_SCAN. After the first call, LIB\$FILE\_SCAN maintains this longword. You must not change the value of **context** in subsequent calls to LIB\$FILE\_SCAN.

Name blocks and file specification strings are allocated by LIB\$FILE\_SCAN, and **context** is used to retain their addresses so they may be deallocated later. If the **context** argument is not passed, unspecified portions of the file specification will be inherited from the previous file specification processed, rather than from multiple input file specifications.

---

## **DESCRIPTION**

LIB\$FILE\_SCAN is called with the address of a File Access Block (FAB) and calls an action routine for each file found and/or error returned. LIB\$FILE\_SCAN allows the search sequence to continue even if an error occurs while processing a particular file.

If this routine is called once for each file specification argument in a command line, portions of the file specifications which are not specified by the user are inherited from the last files processed.

You must call LIB\$FILE\_SCAN\_END before initiating a new sequence of calls to LIB\$FILE\_SCAN.

---

## **CONDITION VALUES RETURNED**

Any condition value returned by the Record Management Service (RMS), Parse.

---

## LIB\$FILE\_SCAN\_END End-of-File Scan

The End-of-File Scan routine is called after each sequence of calls to LIB\$FILE\_SCAN. LIB\$FILE\_SCAN\_END deallocates any saved Record Management Service (RMS) context and/or deallocates the virtual memory that had been allocated for holding the related file specification information.

---

**FORMAT**            **LIB\$FILE\_SCAN\_END** [*fab*] [,*context*]

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        ***fab***  
                           VMS usage: **fab**  
                           type:        **unspecified**  
                           access:      **modify**  
                           mechanism: **by reference**

File access block (FAB) used with LIB\$FILE\_SCAN. The optional **fab** argument is the address of the FAB that contains the address and length of the file specification.

***context***  
                           VMS usage: **context**  
                           type:        **longword (unsigned)**  
                           access:      **modify**  
                           mechanism: **by reference**

Temporary default context used in LIB\$FILE\_SCAN. The optional **context** argument is the address of a longword containing this temporary default context.

---

**DESCRIPTION**      Your program should call LIB\$FILE\_SCAN\_END after each sequence of calls to LIB\$FILE\_SCAN. The function that LIB\$FILE\_SCAN\_END performs depends upon the arguments you specify. If you specify **fab**, LIB\$FILE\_SCAN\_END parses the null string to deallocate any saved RMS context. If you specify **context**, LIB\$FILE\_SCAN\_END deallocates any virtual memory that was allocated for holding the related file specification information. If you specify both **fab** and **context**, LIB\$FILE\_SCAN\_END performs both functions. However, if you do not specify either argument, LIB\$FILE\_SCAN\_END does nothing.

If LIB\$FILE\_SCAN is directed to process the specifications for multiple input files, LIB\$FILE\_SCAN\_END is used to deallocate those saved file specifications. If LIB\$FILE\_SCAN\_END is called by your program after each sequence of calls to LIB\$FILE\_SCAN, it will prevent the defaults from the previous call from affecting context value in the next call to LIB\$FILE\_SCAN.

# LIB\$FILE\_SCAN\_END

LIB\$FILE\_SCAN\_END does this by replacing the context value passed to it with a temporary context value that your program passes to LIB\$FILE\_SCAN the next time it is called.

---

## CONDITION VALUES RETURNED

RMS\$\_NORMAL  
RMS\$\_FAB

Normal successful completion.  
The **fab** argument is not the address of a valid FAB.

---

## LIB\$FIND\_FILE Find File

The Find File routine is called with a wildcard file specification for which it searches. LIB\$FIND\_FILE returns all file specifications that satisfy that wildcard file specification.

---

<b>FORMAT</b>	<b>LIB\$FIND_FILE</b> <i>filespec ,resultant-filespec ,context</i> <i>[ ,default-filespec ] [ ,related-filespec ]</i> <i>[ ,status-value ] [ ,flags ]</i>
---------------	---

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>filespec</i></b> VMS usage: <b>char_string</b> type: <b>character string</b> access: <b>read only</b> mechanism: <b>by descriptor</b>
------------------	---

File specification, which may contain wildcards, that LIB\$FIND\_FILE uses to search for the desired file. The **filespec** argument is the address of a descriptor pointing to the file specification. The maximum length of a file specification is 255 bytes.

The file specification used may also contain a search list logical name. If present, the search list logical name elements can be used as accumulative to related file specifications, so that portions of file specifications not specified by the user will be inherited from previous file specifications.

***resultant-filespec***  
VMS usage: **char\_string**  
type: **character string**  
access: **modify**  
mechanism: **by descriptor**

Resultant file specification that LIB\$FIND\_FILE returns when it finds a file that matches the specification in the **filespec** argument. The **resultant-filespec** argument is the address of a descriptor pointing to the resultant file specification.

***context***  
VMS usage: **context**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

Zero or an address of an internal FAB/NAM buffer from a previous call to LIB\$FIND\_FILE. The **context** argument is an unsigned longword integer containing the address of the context. LIB\$FIND\_FILE uses this argument

# LIB\$FIND\_FILE

to retain the context when processing multiple input files. Portions of file specifications that the user does not specify are inherited from the last files processed because the file contexts are retained in this argument.

## ***default-filespec***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Default file specification. The **default-filespec** argument is the address of a descriptor pointing to the default file specification.

## ***related-filespec***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Related file specification containing the context of the last file processed. The **related-filespec** argument is the address of a descriptor pointing to the related file specification.

## ***status-value***

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by reference**

Record Management Service (RMS) secondary status value from a failing RMS operation. The **status-value** argument is an unsigned longword containing the address of a longword-length buffer to receive the RMS secondary status value (usually returned in the file access block field, FAB\$L\_STV).

## ***flags***

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

User flags. The **flags** argument is the address of an unsigned longword containing the user flags.

The flag bits and their corresponding symbols are described in the following table.

Bit	Symbol	Description
0	NOWILD	If set, LIB\$FIND_FILE returns an error if a wildcard character is input.
1	MULTIPLE	If set, this performs temporary defaulting for multiple input files and the <b>related-filespec</b> argument is ignored. See description of <b>context</b> in LIB\$FILE_SCAN. Each time LIB\$FIND_FILE is called with a different file specification, the specification from the previous call is automatically used as a related file specification. This allows parsing of the elements of a search-list logical name such as DISK2:[SMITH] FIL1.TYP,FIL*2.TYP, and so on. Use of this feature is required to get the desired defaulting with search list logical name. LIB\$FIND_FILE_END must be called between each command line in interactive use or the defaults from the previous command line will affect the current file specification.

## DESCRIPTION

LIB\$FIND\_FILE searches for a certain wildcard file specification and returns all file specifications that satisfy that wildcard file specification.

If you make multiple calls to LIB\$FIND\_FILE, be aware of the following behavior:

- If, when making the multiple calls, the NOWILD bit is not set and the file specification does not contain any wildcard characters, LIB\$FIND\_FILE returns the appropriate file name on the first call and the condition value RMS\$\_NMF on the next call.
- If you make the multiple calls with the NOWILD bit set and the same nonwildcard file specification, LIB\$FIND\_FILE returns the file name on the first call as well as each subsequent call.

You must call LIB\$FIND\_FILE\_END before initiating a new sequence of calls to LIB\$FIND\_FILE.

If the error RMS\$\_CHN is returned, RMS has no more channels to assign. There are two possible reasons for this:

- 1 You did not call LIB\$FIND\_FILE\_END before initiating a new call with a context variable to LIB\$FIND\_FILE. (This is the most common reason.)
- 2 The SYSGEN parameter CHANNELCNT is too low.



# LIB\$FIND\_FILE

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Normal successful completion.

LIB\$\_NOWILD

A wildcard character was present in the file specification parsed and the wildcard flag bit was set to no wildcard. (This is actually the SHR\$\_NOWILD error message after application of the LIB\$ facility code.)

RMS\$\_CHN

No more channels.

RMS\$\_NMF

No more files.

Any condition value returned by RMS Parse and Search services, LIB\$GET\_VM, LIB\$FREE\_VM, or LIB\$COPY\_R\_DX.

---

## LIB\$FIND\_FILE\_END End of Find File

The End of Find File routine is called once after each sequence of calls to LIB\$FIND\_FILE. LIB\$FIND\_FILE\_END deallocates any saved Record Management Service (RMS) context and deallocates the virtual memory used to hold the allocated context block.

---

**FORMAT**            **LIB\$FIND\_FILE\_END**    *context*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        **context**  
                           VMS usage: **context**  
                           type:        **longword (unsigned)**  
                           access:     **read only**  
                           mechanism: **by reference**

Zero or the address of a FAB/NAM buffer from a previous call to LIB\$FIND\_FILE. The **context** argument is the address of a longword that contains this context.

---

**DESCRIPTION**      LIB\$FIND\_FILE\_END should be called by your program after each sequence of calls to LIB\$FIND\_FILE. This will prevent the default values from the previous call from affecting the next file specification.

LIB\$FIND\_FILE\_END deallocates the context used in the last call to LIB\$FIND\_FILE so that the context retained will not be used in subsequent calls to LIB\$FIND\_FILE. If LIB\$FIND\_FILE was directed to process file specifications for multiple input files, the saved file specifications are also deallocated.

---

**CONDITION  
VALUES  
RETURNED**

RMS\$\_NORMAL  
RMS\$\_FAB

Routine successfully completed.  
File access block argument is not the address of a valid FAB.



# LIB\$FIND\_IMAGE\_SYMBOL

## *symbol-value*

VMS usage: **longword\_signed**  
type: **longword (signed)**  
access: **write only**  
mechanism: **by reference**

Symbol value that LIB\$FIND\_IMAGE\_SYMBOL has located. The **symbol-value** argument is the address of a signed longword integer into which LIB\$FIND\_IMAGE\_SYMBOL returns the symbol value. If the symbol is relocatable, the starting virtual address of the shareable image in memory will be added to the symbol value.

## *image-name*

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Default file specification applied to the image name. The optional **image-name** argument is the address of a descriptor pointing to the image name string. If **image-name** is not supplied, then a default file specification of SYS\$SHARE:.EXE is applied to the image name.

---

## DESCRIPTION

The shareable image that LIB\$FIND\_IMAGE\_SYMBOL activates must have been already linked and must be position independent. You must have read access to the shareable image file to use this routine.

LIB\$FIND\_IMAGE\_SYMBOL locates the universal symbol in its database without first processing the **filename** argument. Due to this fact, a reference to a lexically different file name causes a new copy of the same shareable image to be loaded and searched. To avoid this situation, *always specify the desired file name in the same form.*

LIB\$FIND\_IMAGE\_SYMBOL writes the symbol value that it has located into the **symbol-value** argument.

After the first call to LIB\$FIND\_IMAGE\_SYMBOL for a particular image, successive calls for that image will be processed quickly. The image is activated only once and an in-memory database is maintained. There is no way to deallocate this database, nor is there any supported method to remove an activated image from the address space. All images are activated into P0 space.

LIB\$FIND\_IMAGE\_SYMBOL disables AST recognition while it is executing. AST recognition is reenabled before returning to the caller only if AST recognition was previously enabled.

LIB\$FIND\_IMAGE\_SYMBOL signals all errors and returns the status in R0.

# LIB\$FIND\_IMAGE\_SYMBOL

---

## CONDITION VALUES RETURNED

LIB\$_BADCCC	Illegal compilation code.
LIB\$_EOMERROR	Compilation errors.
LIB\$_EOMFATAL	Fatal compilation errors.
LIB\$_EOMWARN	Compilation warnings.
LIB\$_GSDTYP	Illegal universal symbol directory record type.
LIB\$_ILLFMLCNT	Maximum argument count exceeds maximum for routine.
LIB\$_ILLMODNAM	Illegal module name length.
LIB\$_ILLPSCLN	Illegal program section length.
LIB\$_ILLRECLN	Illegal record length in module.
LIB\$_ILLRECLN2	Illegal record length.
LIB\$_ILLRECTYP	Illegal record type in module.
LIB\$_ILLRECTY2	Illegal record type.
LIB\$_ILLSYMLN	Illegal symbol length.
LIB\$_NOEOM	No end of module record contained in the module.
LIB\$_RECTOOSML	Record too small; data overflows object record in module.
LIB\$_SEQUENCE	Illegal record sequence in module.
LIB\$_SEQUENCE2	Illegal record sequence.
LIB\$_STRVL	Illegal object language structure level in module.
Note that all of the above error messages indicate a format error in the shareable image.	
LIB\$_INSVIRMEM	Insufficient virtual memory.
SS\$_IVLOGNAM	The <b>filename</b> argument contained more than just a file name; a device or directory specification was found in the string.

Any condition values returned by LIB\$INSERT\_TREE.

Any condition values returned by RMS.

---

## LIB\$FIND\_VM\_ZONE Return the Next Valid Zone Identifier

The Return the Next Valid Zone Identifier routine returns the zone identifier of the next valid zone in the heap management database.

---

**FORMAT**            **LIB\$FIND\_VM\_ZONE**    *context ,zone-id*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        **context**  
                           VMS usage: **context**  
                           type:        **longword (unsigned)**  
                           access:     **modify**  
                           mechanism: **by reference**

Context specifier. The **context** argument is the address of an unsigned longword used to keep the scan context for finding the next valid zone. **Context** must be 0 to initialize the scan and to start with the first returnable zone identifier.

**zone-id**  
 VMS usage: **identifier**  
 type:        **longword (unsigned)**  
 access:     **write only**  
 mechanism: **by reference**

Zone identifier. The **zone-id** argument is the address of an unsigned longword that receives the zone identifier for the next zone.

---

**DESCRIPTION**        At each call, LIB\$FIND\_VM\_ZONE scans the heap management zone database and returns the **zone-id** of the next valid zone. (The first and second calls to LIB\$FIND\_VM\_ZONE return the **zone-id** of the default zone and string zone, respectively.) This capability allows a program to deal with each VM zone created during the invocation, including those created outside of the program.

The **context** argument controls the state of the scan. It determines what zone to return (the first, the next, and so forth). On the initial call, specified by **context=0**, LIB\$VERIFY\_VM\_ZONE is called to verify the heap management zone database. If the database is corrupt, further calls to this routine will produce no additional useful output.

When no more zones can be found, the routine returns the condition value LIB\$\_NOTFOU.

# LIB\$FIND\_VM\_ZONE

If a zone has been corrupted in some major way (for example, if the validity code has been changed), then this routine may not be able to locate it in the zone database.

Note that ASTs may be disabled while LIB\$FIND\_VM\_ZONE is executing code that depends on the stability of the heap management zone database. In general it is the caller's responsibility to ensure that the calling program has exclusive access to the zone database while scanning for multiple zones with this routine. Results are unpredictable if another thread of control modifies the zone database or the associated areas during the scanning.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_BADZONE	Invalid zone.
LIB\$_NOTFOU	Zone identifier not found (alternate success status).
LIB\$_WRONUMARG	Wrong number of arguments.

---

## EXAMPLE

```
IMPLICIT NONE
INTEGER*4 status,context,zone_id
INTEGER*4 lib$find_vm_zone,lib$show_vm_zone

context = 0
status = lib$find_vm_zone (context, zone_id)
DO WHILE (status)
  print *
  status = lib$show_vm_zone (zone_id, 0)
  status = lib$find_vm_zone (context, zone_id)
END DO
END
```

Sample output for this FORTRAN program is illustrated below.

```
ZONE_ID = 0001B858,  ZONE_NAME = "DEFAULT_ZONE"
```

---

## LIB\$FIXUP\_FLT Fix Floating Reserved Operand

The Fix Floating Reserved Operand routine finds the reserved operand of any F-floating, D-floating, G-floating, or H-floating instruction (with some exceptions) after a reserved operand fault has been signaled. LIB\$FIXUP\_FLT changes the reserved operand from -0.0 to the value of the **new-operand** argument, if present; or to +0.0 if **new-operand** is absent.

---

<b>FORMAT</b>	LIB\$FIXUP_FLT <i>signal-arguments</i> <i>,mechanism-arguments</i> <i>[,new-operand]</i>
---------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>signal-arguments</i></b> VMS usage: <b>vector_longword_unsigned</b> type: <b>unspecified</b> access: <b>read only</b> mechanism: <b>by reference, array reference</b>
------------------	---

Signal argument vector. The **signal-arguments** argument is the address of an array of unsigned longwords containing the signal argument vector.

***mechanism-arguments***  
VMS usage: **vector\_longword\_unsigned**  
type: **unspecified**  
access: **read only**  
mechanism: **by reference, array reference**

Mechanism argument vector. The **mechanism-arguments** argument is the address of an array of unsigned longwords containing the mechanism argument vector.

***new-operand***  
VMS usage: **floating-point**  
type: **F\_floating**  
access: **read only**  
mechanism: **by reference**

An F-floating value to replace the reserved operand. The **new-operand** argument is the address of an F-floating number containing the new operand. This is an optional argument. If omitted, the default value is +0.0.



# LIB\$FIXUP\_FLT

---

## DESCRIPTION

LIB\$FIXUP\_FLT finds the reserved operand of any F-floating, D-floating, G-floating, or H-floating instruction (with some exceptions) after a reserved operand fault has been signaled. LIB\$FIXUP\_FLT changes the reserved operand from -0.0 to the value of the **new-operand** argument, if present; or to +0.0 if **new-operand** is absent. LIB\$FIXUP\_FLT cannot handle the following cases and will return a status of SS\$\_RESIGNAL if any of them occur:

- The currently active signaled condition is not SS\$\_ROPRAND.
- The reserved operand's data type is not F-floating, D-floating, G-floating, or H-floating.
- The reserved operand is an element in a POLYx coefficient table.

If the status value returned from LIB\$FIXUP\_FLT is seen by the condition handling facility (as would be the case if LIB\$FIXUP\_FLT was the handler), any success value is equivalent to SS\$\_CONTINUE, which causes the instruction to be restarted. Any failure value is equivalent to SS\$\_RESIGNAL, which causes the condition to be resignaled to the next handler. This resignal status is because the condition handler (LIB\$FIXUP\_FLT) was unable to handle the condition correctly.

LIB\$FIXUP\_FLT can be enabled directly as a condition handler. The **signal-arguments** and **mechanism-arguments** arguments are passed to the condition handler by VMS exception dispatching.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. The reserved operand was found and has been fixed.
SS\$_ACCVIO	Access violation. An argument to LIB\$FIXUP_FLT or an operand of the faulting instruction could not be read or written.
SS\$_RESIGNAL	The signaled condition was not SS\$_ROPRAND, or the reserved operand was not a floating-point value or was an element in a POLYx table.
SS\$_ROPRAND	Reserved operand fault. The optional argument <b>new-operand</b> was supplied but was itself an F-floating reserved operand.
LIB\$_BADSTA	Bad stack. The stack frame linkage has been corrupted since the time of the reserved operand exception.

---

## LIB\$FLT\_UNDER Floating-Point Underflow Detection

The Floating-Point Underflow Detection routine enables or disables floating-point underflow detection for the calling routine activation. The previous setting is returned as a function value.

---

**FORMAT**                    **LIB\$FLT\_UNDER**    *new-setting*

---

**RETURNS**                    VMS usage: **longword\_unsigned**  
                                   type:            **longword (unsigned)**  
                                   access:        **write only**  
                                   mechanism:    **by value**

The old floating-point underflow enable setting (the previous contents of the SF\$W\_PSW[PSW\$V\_FU] in the caller's frame).

---

**ARGUMENT**                    ***new-setting***  
                                   VMS usage: **longword\_unsigned**  
                                   type:            **longword (unsigned)**  
                                   access:        **read only**  
                                   mechanism:    **by reference**

New floating-point underflow enable setting. The **new-setting** argument is the address of an unsigned byte containing the new setting. Bit 0 set to 1 means enable; bit 0 set to 0 means disable.

---

**DESCRIPTION**                LIB\$FLT\_UNDER affects only the current routine activation and does not affect any of its callers or any routines that it may call. However, the setting does remain in effect for any routines entered through a JSB entry point.

The caller's stack frame will be modified by this routine.

---

**CONDITION  
VALUES  
RETURNED**                    *None.*

# LIB\$FLT\_UNDER

---

## EXAMPLE

C+

C This FORTRAN example program illustrates

C the use of LIB\$FLT\_UNDER.

C-

```
      INTEGER*4  NEW_SETTING
      REAL*4    X , Y , Z

      NEW_SETTING = 0
      X = 1E-20
      Y = 1E20

      CALL LIB$FLT_UNDER( NEW_SETTING )

      TYPE *, 'First Case: This should not have an underflow exception'

      Z = X / Y

      TYPE *, 'If this lines prints then the underflow exception
1 was disabled.'
      TYPE *

      NEW_SETTING = 1
      X = 1E-20
      Y = 1E20

      CALL LIB$FLT_UNDER( NEW_SETTING )

      TYPE * , 'Second Case: This should have an underflow exception
1 and then stop.'

      Z = X / Y

      TYPE * , 'If this line prints, then the underflow exception
1 was disabled.'

c      CALL EXIT

      END
```

In this FORTRAN example, floating-point underflow detection is disabled the first time X is divided by Y. The second time, underflow detection is enabled, and the program stops because of the error generated.

---

## LIB\$FORMAT\_DATE\_TIME Format Date and/or Time

The Format Date and/or Time routine allows the user to select at run time a specific output language and format for a date or time, or both.

---

<b>FORMAT</b>	<b>LIB\$FORMAT_DATE_TIME</b>	<i>date-string</i> [, <i>date</i> ] [, <i>user-context</i> ] [, <i>date-length</i> ] [, <i>flags</i> ]
---------------	------------------------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>date-string</i></b> VMS usage: <b>char_string</b> type: <b>character string</b> access: <b>write only</b> mechanism: <b>by descriptor</b>
------------------	---

Receives the requested date or time, or both, that has been formatted for output according to the currently selected format and language. The **date-string** argument is the address of a descriptor pointing to this string.

### ***date***

VMS usage: **date\_time**  
type: **quadword (unsigned)**  
access: **read only**  
mechanism: **by reference**

The date or time, or both, to be formatted for output. The **date** argument is the address of an unsigned quadword that contains the absolute date or time, or both to be formatted. If you omit this argument, or if you supply a zero passed by value, then the current system time is used. Note that the **date** argument must represent an absolute time, not a delta time.

### ***user-context***

VMS usage: **context**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

User context that retains the translation context over multiple calls to this routine. The **user-context** argument is the address of an unsigned longword that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.

# LIB\$FORMAT\_DATE\_TIME

The **user-context** parameter is optional. However, if a context cell is not passed, the routine LIB\$FORMAT\_DATE\_TIME may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, DIGITAL recommends that users specify a different context cell for each calling thread.

## ***date-length***

VMS usage: **longword\_signed**  
type: **longword (signed)**  
access: **write only**  
mechanism: **by reference**

Number of bytes of text written to the **date-string** argument. The **date-length** argument is the address of a signed longword that receives this string length. Note that **date-length** specifies the number of bytes of text, not the number of characters, written to **date-string**.

## ***flags***

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Bit mask that allows the user to specify whether the date, time, or both are output. The **flags** argument is the address of an unsigned bit mask containing the specified values. Valid values are LIB\$M\_DATE\_FIELDS and LIB\$M\_TIME\_FIELDS.

Default values are determined as follows:

- If the **flags** argument is omitted, LIB\$FORMAT\_DATE\_TIME determines which fields to format according to the current definition of LIB\$DT\_FORMAT.
- If the **flags** argument is specified, LIB\$FORMAT\_DATE\_TIME uses the **flags** value to determine which fields to format. That is, the **flags** argument can be used to override the definition of LIB\$DT\_FORMAT when specifying which fields should be formatted for output. If the field specified by **flags** was not assigned a format through the definition of LIB\$DT\_FORMAT, the standard VMS format is used.

---

## DESCRIPTION

The LIB\$FORMAT\_DATE\_TIME routine formats a VMS internal format date-time quadword into a textual string of some predefined format. The language to be used and the format in which to output the information are programmable using either of the following methods.

- The language and format are programmable at compile time through the use of the routine LIB\$INIT\_DATE\_TIME\_CONTEXT.
- The language and format are determined at run time through the translation of the logical names SYS\$LANGUAGE and LIB\$DT\_FORMAT.

In general, if an application is formatting text for internal storage or transmission, the language and format should be specified at compile time. If this is the case, use the routine LIB\$INIT\_DATE\_TIME\_CONTEXT to specify the language and format of your choice.

# LIB\$FORMAT\_DATE\_TIME

If an application is formatting text for presentation to a user, the logical name method of specifying language and format should be used. In this method, the user assigns equivalence names to the logical names SYS\$LANGUAGE and LIB\$DT\_FORMAT, thereby selecting the language and format of the date and time at run time.

If the logical name method is used, the translations of the logical names SYS\$LANGUAGE and LIB\$DT\_FORMAT specify one or more executive mode logicals, which in turn must be translated to determine the actual format string. These additional logicals supply such things as the names of the days of the week and the months in the selected language (determined by SYS\$LANGUAGE). All of these logicals are predefined, so that a non-privileged user can select any one of these languages and formats. A user can create his or her own languages and formats; however, the CMEXEC, SYSNAME, and SYSPRV privileges are required.

With the exception of SYS\$LANGUAGE and LIB\$DT\_FORMAT, all logical names used by this routine must be defined from the executive mode.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_ENGLUSED	English used; unable to determine or use the specified language.
LIB\$_DEFFORUSE	Default format used; unable to determine the desired format.
LIB\$_UNRFORCOD	Unrecognized format code.
LIB\$_STRTRU	Output string truncated.
LIB\$_ABSTIMREQ	Absolute time required.
LIB\$_REENTRANCY	Reentrant invocation with same context variable.

Any condition values returned by SYS\$NUMTIM, LIB\$GET\_VM, and LIB\$ANALYZE\_SDESC.

# LIB\$FREE\_DATE\_TIME\_CONTEXT

---

**LIB\$FREE\_DATE\_TIME\_CONTEXT** Free the Context Area Used When Formatting Dates and Times for Input or Output

The Free the Context Area Used When Formatting Dates and Times for Input or Output routine frees the virtual memory associated with the context area used by the date/time input and output Formatting Routines.

---

**FORMAT** LIB\$FREE\_DATE\_TIME\_CONTEXT [*user-context*]

---

**RETURNS** VMS usage: **cond\_value**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by value**

---

**ARGUMENTS** *user-context*  
VMS usage: **context**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

User context that retains the translation context over multiple calls to the date/time input and output Formatting Routines. The **user-context** argument is the address of an unsigned longword that contains this context. If the **user-context** argument was not specified in the call to LIB\$FORMAT\_DATE\_TIME, LIB\$CONVERT\_DATE\_STRING, or LIB\$GET\_MAXIMUM\_DATE\_LENGTH, then no argument should be supplied when calling this routine.

---

**DESCRIPTION** The LIB\$FREE\_DATE\_TIME\_CONTEXT routine frees the virtual memory associated with the context area used by the date/time input and output formatting routines. A call to this routine is optional, since the same functions will be performed at image exit.

# LIB\$FREE\_DATE\_TIME\_CONTEXT

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Normal successful completion.

Any condition value returned by LIB\$FREE\_VM. If one of these condition values is returned, it indicates either an internal coding error or memory that was corrupted by the user's program.



# LIB\$FREE\_EF

---

## LIB\$FREE\_EF Free Event Flag

The Free Event Flag routine frees a local event flag previously allocated by LIB\$GET\_EF. LIB\$FREE\_EF is the complement of LIB\$GET\_EF.

---

**FORMAT**            **LIB\$FREE\_EF** *event-flag-number*

---

**RETURNS**            VMS usage: **cond\_value**  
                         type:        **longword (unsigned)**  
                         access:     **write only**  
                         mechanism: **by value**

---

**ARGUMENT**            ***event-flag-number***  
                         VMS usage: **ef\_number**  
                         type:        **longword integer (unsigned)**  
                         access:     **read only**  
                         mechanism: **by reference**

Event flag number to be deallocated by LIB\$FREE\_EF. The **event-flag-number** argument is the address of a signed longword integer that contains the event flag number, which is the value returned to the user by LIB\$GET\_EF.

---

**DESCRIPTION**        When a local event flag allocated by calling LIB\$GET\_EF is no longer needed, LIB\$FREE\_EF should be called to free the event flag for use by other routines.

---

<b>CONDITION VALUES RETURNED</b>	SS\$_NORMAL	Routine successfully completed.
	LIB\$_EF_ALRFRE	Event flag already free.
	LIB\$_EF_RESSYS	Event flag reserved to system. This error occurs if the event flag number is outside the ranges of 1 to 23 and 32 to 63.

---

### EXAMPLE

For an example of using LIB\$FREE\_EF, see the example under LIB\$GET\_EF.

---

## LIB\$FREE\_LUN Free Logical Unit Number

The Free Logical Unit Number routine releases a logical unit number allocated by LIB\$GET\_LUN to the pool of available numbers. LIB\$FREE\_LUN is the complement of LIB\$GET\_LUN.

---

**FORMAT**            **LIB\$FREE\_LUN**    *logical-unit-number*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENT**            *logical-unit-number*  
                           VMS usage: **longword\_signed**  
                           type:            **longword integer (signed)**  
                           access:        **read only**  
                           mechanism:    **by reference**

Logical unit number to be deallocated. The **logical-unit-number** argument is the address of a signed longword integer that contains this logical unit number, which is the value previously returned by LIB\$GET\_LUN.

---

**DESCRIPTION**        When a logical unit number allocated by calling LIB\$GET\_LUN is no longer needed, it should be released for use by other routines.

This routine is useful only in BASIC or FORTRAN programs.

---

<b>CONDITION VALUES RETURNED</b>	SS\$_NORMAL	Routine successfully completed.
	LIB\$_LUNALRFRE	Logical unit number is already free.
	LIB\$_LUNRESSYS	Logical unit number reserved to system. This occurs if the specified logical unit number is outside the range of 100 to 119.

# LIB\$FREE\_TIMER

---

## LIB\$FREE\_TIMER Free Timer Storage

The Free Timer Storage routine frees the storage allocated by LIB\$INIT\_TIMER.

---

**FORMAT**            **LIB\$FREE\_TIMER** *handle-address*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENT**            *handle-address*  
                          VMS usage: **address**  
                          type:        **longword (unsigned)**  
                          access:     **modify**  
                          mechanism: **by reference**

Pointer to a block of storage containing the value returned by a previous call to LIB\$INIT\_TIMER; this is the storage that LIB\$FREE\_TIMER deallocates. The **handle-address** argument is the address of an unsigned longword containing that value.

---

**DESCRIPTION**        LIB\$FREE\_TIMER frees a block of storage previously allocated by LIB\$INIT\_TIMER. LIB\$FREE\_TIMER assumes that **handle-address** was returned by a previous call to LIB\$INIT\_TIMER. If the block referred to by **handle-address** was not allocated by LIB\$INIT\_TIMER, LIB\$FREE\_TIMER returns an error. If the routine completes successfully, LIB\$FREE\_TIMER sets **handle-address** to zero.

---

<b>CONDITION VALUES RETURNED</b>	SS\$_NORMAL	Routine successfully completed.
	LIB\$_INVARG	Invalid argument; <b>handle-address</b> was not supplied or did not point to a timer block.
	LIB\$_BADBLOADR	Bad block address; LIB\$FREE_VM could not deallocate the block to which <b>handle-address</b> points.



# LIB\$FREE\_VM

You must specify the same **zone-id** value as when you called LIB\$GET\_VM to allocate the block. An error status will be returned if you specify an incorrect **zone-id**. The **zone-id** argument is optional. If **zone-id** is omitted or if the longword contains the value zero, LIB\$VM's default zone is used.

---

## DESCRIPTION

LIB\$FREE\_VM returns the block of memory to a free list associated with the zone, so the block is available on a subsequent call to LIB\$GET\_VM for the zone.

The **base-address** argument must contain the address of the first byte of memory that was allocated by a previous call to LIB\$GET\_VM. LIB\$FREE\_VM rounds up the value of **number-of-bytes** to a multiple of the block size for the zone.

**Note:** You cannot free part of a block that was allocated by a call to LIB\$GET\_VM. The whole block must be freed by a single call to LIB\$FREE\_VM.

Neither can you combine contiguous blocks of memory that were allocated by several calls to LIB\$GET\_VM into one larger block that is freed by a single call to LIB\$FREE\_VM.

If you specified deallocation filling when you created the zone, LIB\$FREE\_VM will fill each byte freed. Note that part of a free block is used to store control information, so some bytes will not contain the fill value.

LIB\$FREE\_VM is fully reentrant, so it can be called by routines executing at AST-level or in an Ada multitasking environment.

If the zone you are freeing was created using the LIB\$CREATE\_USER\_VM\_ZONE routine, then you must have an appropriate action routine for the free operation. That is, in your call to LIB\$CREATE\_USER\_VM\_ZONE, you must have specified a user deallocation procedure.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_BADBLOADR	<b>Base-address</b> contained a bad block address. An address was outside of the area allocated by LIB\$GET_VM, or the contents of <b>base-address</b> were not properly aligned, or part of the space being deallocated was previously deallocated.
LIB\$_BADBLOSIZ	<b>Number-of-bytes</b> is less than or equal to 0, or the <b>number-of-bytes</b> argument is incorrect for a zone containing fixed size blocks.

---

## LIB\$FREE\_VM\_PAGE Free Virtual Memory Page

The Free Virtual Memory Page routine deallocates a block of contiguous pages that were allocated by previous calls to LIB\$GET\_VM\_PAGE.

---

**FORMAT**            **LIB\$FREE\_VM\_PAGE** *number-of-pages ,base-address*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        ***number-of-pages***  
                           VMS usage: **longword\_signed**  
                           type:            **longword integer (signed)**  
                           access:        **read only**  
                           mechanism:    **by reference**

Number of pages. The **number-of-pages** argument is the address of a longword integer which specifies the number of contiguous pages to be deallocated. The value of **number-of-pages** must be greater than zero.

***base-address***

VMS usage: **address**  
 type:            **longword (unsigned)**  
 access:        **read only**  
 mechanism:    **by reference**

Block address. The **base-address** argument is the address of a longword which contains the address of the first byte of the first page to be deallocated.

---

**DESCRIPTION**      LIB\$FREE\_VM\_PAGE deallocates a block of contiguous pages starting at **base-address**. Each of the pages specified by **number-of-pages** and **base-address** must have been allocated by previous calls to LIB\$GET\_VM\_PAGE. The pages are returned to the processwide page pool and are available to satisfy subsequent calls to LIB\$GET\_VM\_PAGE.

You can free a smaller group of pages than you allocated. That is, if you allocated a group of contiguous pages by a single call to LIB\$GET\_VM\_PAGE, you can deallocate those pages in several calls to LIB\$FREE\_VM\_PAGE. You can also combine contiguous groups of pages that were allocated in several calls to LIB\$GET\_VM\_PAGE into one large group that is freed by a single call to LIB\$FREE\_VM\_PAGE.

LIB\$FREE\_VM\_PAGE is fully reentrant, so it may be called by routines executing at AST level or in an Ada multitasking environment.

# LIB\$FREE\_VM\_PAGE

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Normal successful completion.

LIB\$\_BADBLOADR

Pages not allocated by LIB\$GET\_VM\_PAGE, the value of **base-address** is not a page boundary, or the pages were previously freed.

LIB\$\_BADBLOSIZ

**Number-of-pages** is less than or equal to zero.





# LIB\$GETDVI

The **device-name** may be either a physical device name or a logical name. If the first character in the string is an underscore character (`_`), the name is considered a physical device name. Otherwise, the name is considered a logical name, and logical name translation is performed until either a physical device name is found or the system default number of translations has been performed.

If **device-name** is not specified, **channel** is used instead. You must specify either **channel** or **device-name**, but not both. If neither is specified, the error status `SS$_IVDEVNAM` is returned. The device name must not be longer than 255 characters.

## ***longword-integer-value***

VMS usage: **longword\_signed**  
type: **longword (signed)**  
access: **write only**  
mechanism: **by reference**

Numeric value of the information requested. The **longword-integer-value** argument is the address of a signed longword containing the numeric value. If an item is listed as only returning a string value, this argument is ignored.

## ***resultant-string***

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

String representation of the information requested. The **resultant-string** argument is the address of a descriptor pointing to this information. If **resultant-string** is not specified and if the value returned has only a string representation, the error status `LIB$_INVARG` is returned.

Refer to Table LIB-4 for a description of the string representation used for each item.

## ***resultant-length***

VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **write only**  
mechanism: **by reference**

Number of significant characters written to **resultant-string** by LIB\$GETDVI. The **resultant-length** argument is the address of an unsigned word containing this length.

---

## DESCRIPTION

LIB\$GETDVI returns two categories of information.

- Primary device characteristics
- Secondary device characteristics

LIB\$GETDVI does not allow you to get more than one item of information in a single call.

LIB\$GETDVI provides the following features in addition to those provided by the \$GETDVI system service.

- Instead of a list of item descriptors, which may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.
- For items which return numeric values, LIB\$GETDVI can optionally provide a formatted string interpretation of the value. For example, if the device owner UIC is requested, LIB\$GETDVI can return the UIC formatted as [identifier].
- For string arguments, LIB\$GETDVI understands all string classes supported by the Run-Time Library.
- Calls to LIB\$GETDVI are synchronous; LIB\$GETDVI calls LIB\$GET\_EF to allocate a local event flag number for synchronization.

See the description of the \$GETDVI system service in the *VMS System Services Reference Manual* for more detailed information.

## Item Codes

All item codes that can be used with the \$GETDVI system service may be used as the **item-code** argument to LIB\$GETDVI. These codes have symbolic names beginning with DVI\$\_\_.

The use of a DVI\$\_\_ code by itself will return the primary device characteristic associated with that code. To obtain the secondary device characteristics, add 1 to the code. See the description of the \$GETDVI system service for a list of the defined item codes. The symbolic names for these items are defined in DIGITAL-supplied symbol libraries in module \$DVIDEF (where appropriate).

## Value Formats

By using the **longword-integer-value** and **resultant-string** arguments to LIB\$GETDVI, the information requested can be returned in two different fashions.

- For those items described as “string” in the table of Item Identifier Codes for the \$GETDVI service, the value is returned in **resultant-string**.
- For all other items—those that have numeric values—the numeric representation is returned in **longword-integer-value** (if specified), and a formatted string interpretation of the value is returned in **resultant-string**.

Each formatted item is written left-justified; **resultant-length**, if specified, gives the number of characters used. Table LIB-4 lists the formats used for the string interpretations.

# LIB\$GETDVI

**Table LIB-4 Formats Used for LIB\$GETDVI Strings**

<b>Item or Format</b>	<b>Description</b>
DVI\$_ACPPID	The string value is returned as an 8-digit hexadecimal number.
DVI\$_PID	The string value is returned as an 8-digit hexadecimal number.
DVI\$_ACPTYPE	The ACP type string is one of the following: NONE No ACP F11V1 Files-11 Level 1 F11V2 Files-11 Level 2 MTA Magnetic Tape NET Networks REM Remote I/O JNL Journal
DVI\$_OWNUIC	The standard UIC format [group,member] is used. If the format of a UIC includes identifiers from the access rights database in place of the octal group and member numbers, the UIC string returned will have these identifiers, if available.
DVI\$_VPROT	The volume protection string is in the following form: SYSTEM=RWLP,OWNER=RWLP,GROUP=RWLP,WORLD=RWLP If a category has no access, the equal sign is omitted. The string will not contain any embedded spaces.
Boolean	The value string returned is TRUE if the low bit of the value is set, or FALSE if the low bit is clear.
All others	The value string is returned in the form of an unsigned decimal integer.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Normal successful completion.
LIB\$_STRTRU	String truncated. This is an alternate success return status. The fixed-length <b>resultant-string</b> argument could not contain all the characters of the returned item.
SS\$_BADPARAM	Unrecognized item code. <b>Item-code</b> was not recognized as valid by \$GETSYI.
SS\$_IVDEVNAM	The device name string contains invalid characters, or neither the <b>channel</b> nor <b>device-name</b> arguments were specified.
LIB\$_INSEF	Insufficient event flags. A local event flag number could not be allocated by a call to LIB\$GET_EF.
LIB\$_INVARG	Invalid arguments. The \$GETSYI Item Identifier code describes the item as "string", and no <b>resultant-string</b> argument was specified.
LIB\$_INVSTRDES	Invalid string descriptor. The descriptor of the <b>resultant-string</b> argument is not a valid descriptor.
LIB\$_WRONUMARG	Wrong number of arguments. An incorrect number of arguments was passed to LIB\$GETDVI.

Any condition values returned by LIB\$COPY\_XXX.

Any condition values returned by SYS\$GETDVI.



A 1- to 15-character string specifying the name of the process for which you are requesting information. The **process-name** argument is the address of a descriptor pointing to the process name string. The name must correspond exactly to the name of the process for which you are requesting information; LIB\$GETJPI does not allow trailing blanks or abbreviations.

If you do not specify **process-name**, **process-id** is used. If you specify neither **process-name** or **process-id**, the caller's process is used.

### ***resultant-value***

VMS usage: **varying\_arg**  
 type: **unspecified**  
 access: **write only**  
 mechanism: **by reference**

Numeric value of the information you request. The **resultant-value** argument is the address of a longword or quadword into which LIB\$GETJPI writes the numeric value of this information. Refer to Table LIB-5 for information on which items return longword values and which return quadword values. If the item you request returns only a string value, this argument is ignored.

### ***resultant-string***

VMS usage: **char\_string**  
 type: **character string**  
 access: **write only**  
 mechanism: **by descriptor**

String representation of the information you request. The **resultant-string** argument is the address of a character string into which LIB\$GETJPI writes the string representation. Table LIB-5 describes the string representation used for each item.

If you do not include **resultant-string**, but the item you request has only a string representation, the error status LIB\$\_INVARG is returned.

### ***resultant-length***

VMS usage: **word\_unsigned**  
 type: **word (unsigned)**  
 access: **write only**  
 mechanism: **by reference**

Number of significant characters written to **resultant-string** by LIB\$GETJPI. The **resultant-length** argument is the address of an unsigned word integer into which LIB\$GETJPI writes the number of characters.

---

## **DESCRIPTION**

LIB\$GETJPI provides the following features in addition to those provided by the \$GETJPI system service.

- Instead of a list of item descriptors, which may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.
- For items which return numeric values, LIB\$GETJPI can optionally provide a formatted string interpretation of the value. For example, if the process UIC is requested, LIB\$GETJPI can return the UIC formatted as [g,m].

# LIB\$GETJPI

- For string arguments, all string classes supported by the Run-Time Library are understood.
- Calls to LIB\$GETJPI are synchronous. LIB\$GETJPI calls LIB\$GET\_EF to allocate a local event flag number for synchronization.

See the description of the \$GETJPI system service in the *VMS System Services Reference Manual* for more information.

By using the **resultant-value** and **resultant-string** arguments to LIB\$GETJPI, you can request that the information be returned in two ways. For those items described as "string" in the table of Item Identifier Codes for the \$GETJPI service, the value is returned in **resultant-string**. For all other items—those which have numeric values—the numeric representation is returned in **resultant-value** (if specified), and a formatted string interpretation of the value is returned in **resultant-string**.

Each formatted item is written left-justified; **resultant-length**, if specified, gives the number of characters used. For the items that return blank-padded strings (for example, JPI\$\_USERNAME) trailing blanks are removed.

Table LIB-5 lists the formats used for the string interpretations.

**Table LIB-5 Item Code Formats for LIB\$GETJPI**

Item or Format	Description
JPI\$_AUTHPRIV	The string representation of these quadword privilege masks is a list of each privilege that is enabled. The privilege names are in uppercase, and are separated by commas.
JPI\$_CURPRIV	Same as for JPI\$AUTHPRIV.
JPI\$_IMAGPRIV	Same as for JPI\$AUTHPRIV.
JPI\$_PROCPRIV	Same as for JPI\$AUTHPRIV.
JPI\$_LOGINTIM	The string representation of the quadword time is a standard absolute date-time string.
JPI\$_PID	The process identification string is an 8-digit hexadecimal number.

Table LIB-5 (Cont.) Item Code Formats for LIB\$GETJPI

Item or Format	Description
JPI\$_STATE	The process state string is one of the following: CEF Common event flag wait COM Computable COMO Computable, outswapped CUR Current process COLPG Collided page wait FPG Free page wait HIB Hibernate wait HIBO Hibernate wait, outswapped LEF Local event flag wait LEFO Local event flag wait, outswapped MWAIT Mutex and miscellaneous resource wait PFW Page fault wait SUSP Suspended SUSPO Suspended, outswapped
JPI\$_UIC	The standard UIC format [group,member] is used. If the format of a UIC includes identifiers from the access rights database in place of the octal group and member numbers, the UIC string returned will have these identifiers, if available.
JPI\$_MODE	The current mode string is one of the following: BATCH, INTERACTIVE or NETWORK.
All others	The string value is returned as an unsigned decimal integer.



# LIB\$GETJPI

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_STRTRU	String truncated. This is an alternate success return status. The fixed-length <b>resultant-string</b> argument could not contain all the characters of the returned item.
SS\$_BADPARAM	Unrecognized item code. <b>Item-code</b> was not recognized as valid by \$GETJPI.
LIB\$_INSEF	Insufficient event flags. A local event flag number could not be allocated by a call to LIB\$GET_EF.
LIB\$_INVARG	Invalid arguments. The \$GETSYI Item Identifier code describes the item as "string", and no <b>resultant-string</b> argument was specified.
LIB\$_INVSTRDES	Invalid string descriptor. The descriptor for a string argument was not a valid string descriptor.
LIB\$_WRONUMARG	Wrong number of arguments. An incorrect number of arguments was passed to LIB\$GETJPI.

Any condition value returned by LIB\$SCOPY\_xxx.

Any condition value returned by SYS\$GETJPI.

---

## LIB\$GETQUI Get Queue Information

The Get Queue Information routine provides a simplified interface to the \$GETQUI system service. It provides queue, job, file, characteristic, and form information about a specified process.

LIB\$GETQUI obtains only one item of information in a single call.

---

<b>FORMAT</b>	<b>LIB\$GETQUI</b> <i>function-code</i> [, <i>item-code</i> ] [, <i>search-number</i> ] [, <i>search-name</i> ] [, <i>search-flags</i> ] [, <i>resultant-value</i> ] [, <i>resultant-string</i> ] [, <i>resultant-length</i> ]
---------------	---

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>function-code</i></b> VMS usage: <b>longword_signed</b> type: <b>longword (signed)</b> access: <b>read only</b> mechanism: <b>by reference</b>
------------------	--

Function code specifying the function that LIB\$GETQUI is to perform. The **function-code** argument is the address of a signed longword containing the function code.

LIB\$GETQUI accepts all \$GETQUI function codes. These names begin with QUI\$\_ and are defined in DIGITAL-supplied symbol libraries in module \$QUIDEF.

***item-code***  
VMS usage: **longword\_signed**  
type: **longword (signed)**  
access: **read only**  
mechanism: **by reference**

Item identifier code specifying the item of information you are requesting. The **item-code** argument is the address of a signed longword containing the item code. You may request only one item in each call to LIB\$GETQUI.

LIB\$GETQUI accepts all \$GETQUI item codes. These names begin with QUI\$\_ and are defined in DIGITAL-supplied symbol libraries in module \$QUIDEF.

# LIB\$GETQUI

## ***search-number***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Numeric value used to process your request. The **search-number** argument is the address of a signed longword integer containing the number needed to process your request. **Search-number** directly corresponds to QUI\$\_SEARCH\_NUMBER as described by the \$GETQUI system service.

## ***search-name***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Character string used to process your request. The **search-name** argument is the address of a string descriptor that provides the name needed to process your request. **Search-name** directly corresponds to QUI\$\_SEARCH\_NAME as described by the \$GETQUI system service.

## ***search-flags***

VMS usage: **longword\_unsigned**  
type: **longword integer (unsigned)**  
access: **read only**  
mechanism: **by reference**

Optional bit mask indicating request to be performed. The **search-flags** argument is the address of an unsigned longword integer containing the bit mask. **Search-flags** directly corresponds to \$QUI\$\_SEARCH\_FLAGS as described by the \$GETQUI system service.

## ***resultant-value***

VMS usage: **varying\_arg**  
type: **unspecified**  
access: **write only**  
mechanism: **by reference**

Numeric value of the information you requested. The **resultant-value** argument is the address of a longword, quadword or octaword into which LIB\$GETQUI writes the numeric value of this information. Refer to Table LIB-6 for information on which items return values other than longwords. If the item you requested returns only a string value, this argument is ignored.

## ***resultant-string***

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

String representation of the information you requested. The **resultant-string** argument is the address of a character string into which LIB\$GETQUI writes the string representation. Table LIB-6 describes the string representation used for each item.

If you do not include **resultant-string**, but the item you request has only a string representation, the error status LIB\$\_INVARG is returned.

## **resultant-length**

VMS usage: **word\_signed**  
type: **word integer (signed)**  
access: **write only**  
mechanism: **by reference**

Number of significant characters written to **resultant-string** by LIB\$GETQUI. The **resultant-length** argument is the address of a signed word integer into which LIB\$GETQUI writes the number of characters.

---

## DESCRIPTION

LIB\$GETQUI provides a simplified interface to the \$GETQUI system service. It provides queue, job, file, characteristic, and form information about a specified process. This routine obtains only one item of information in a single call.

LIB\$GETQUI provides the following features in addition to those provided by the \$GETQUI system service.

- Instead of a list of item descriptors that may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.
- For items that return numeric values, LIB\$GETQUI optionally can provide a formatted string interpretation of the value. For example, if you request the characteristics of a queue, LIB\$GETQUI can return the list of characteristics as "23,42,76,98,125".
- For string arguments, all string classes supported by the Run-Time Library are understood.
- Calls to LIB\$GETQUI are synchronous. LIB\$GETQUI calls \$GETQUIW to force the synchronization.

LIB\$GETQUI retains context. This means that previous calls to LIB\$GETQUI affect current calls to LIB\$GETQUI.

See the description of the \$GETQUI system service in the *VMS System Services Reference Manual* for more information.

By using the **resultant-value** and **resultant-string** arguments to LIB\$GETQUI, you can request that the information be returned in two ways. For items that have numeric values, the numeric representation is returned in **resultant-value** (if specified), and a formatted string interpretation of the value is returned in **resultant-string**. For those items described as "string" in the table of Item Identifier Codes for the \$GETQUI service, the value is returned in **resultant-string**.

Each formatted item is written left-justified; **resultant-length**, if specified, gives the number of characters used. For the items that return blank-padded strings, trailing blanks are removed.

The \$GETQUI system service requires some item codes. LIB\$GETQUI provides those item codes for you by corresponding your input to LIB\$GETQUI directly to the required input codes.

# LIB\$GETQUI

The following table describes all of the required and optional input needed to perform your task with LIB\$GETQUI.

Function	Input Description
QUI\$_CANCEL	Accepts no input.
QUI\$_DISPLAY_CHARACTERISTIC	A characteristic name or number, or both. Optionally, a search flags number.
QUI\$_DISPLAY_ENTRY	Optionally, an entry number, user name, and search flags number. The default user name is that of the calling process.
QUI\$_DISPLAY_FILE	Optionally, a search flags number.
QUI\$_DISPLAY_FORM	A form name or number, or both. Optionally, a search flags number.
QUI\$_DISPLAY_JOB	Optionally, a search flags number.
QUI\$_DISPLAY_QUEUE	A queue name. Optionally, a search flags number.
QUI\$_TRANSLATE_QUEUE	A queue name.

Table LIB-6 lists the formats used for the string interpretations.

**Table LIB-6 Item Code Formats for LIB\$GETQUI**

Item or Format	Description
QUI\$_AFTER_TIME	Returns a quadword <b>resultant-value</b> as well as a <b>resultant-string</b> .
QUI\$_CHARACTERISTICS	Returns an octaword <b>resultant-value</b> as well as a comma-separated list that lists all the characteristic numbers, output as a <b>resultant-string</b> .
QUI\$_SUBMISSION_TIME	Returns a quadword <b>resultant-value</b> as well as a <b>resultant-string</b> .
QUI\$_UIC	Returns a formatted <b>resultant-string</b> as well as a longword.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	String truncated. This is an alternate success return status. The fixed-length <b>resultant-string</b> argument could not contain all the characters of the returned item.
SS\$_BADPARAM	Unrecognized item code. <b>Item-code</b> was not recognized as valid by \$GETQUI.
LIB\$_INSEF	Insufficient event flags. A local event flag number could not be allocated by a call to LIB\$GET_EF.
LIB\$_INVARG	Invalid arguments. The \$GETSYI Item Identifier code describes the item as "string", and no <b>resultant-string</b> argument was specified.
LIB\$_INVSTRDES	Invalid string descriptor. The descriptor for a string argument was not a valid string descriptor.
LIB\$_WRONUMARG	Wrong number of arguments. An incorrect number of arguments was passed to LIB\$GETQUI.

Any condition value returned by LIB\$COPY\_XXX.

Any condition value returned by SYS\$GETQUI.



***resultant-length***

VMS usage: **word\_unsigned**  
 type: **word (unsigned)**  
 access: **write only**  
 mechanism: **by reference**

Number of significant characters written to **resultant-string**, not including blank padding or truncated characters. The **resultant-length** argument is the address of an unsigned word into which LIB\$GETSYI returns this number.

***cluster-system-id***

VMS usage: **identifier**  
 type: **longword (unsigned)**  
 access: **modify**  
 mechanism: **by reference**

Cluster system identification (CSID) of the node for which information is to be returned. The **cluster-system-id** argument is the address of this CSID. If **cluster-system-id** is specified and is nonzero, **node-name** is not used. If **cluster-system-id** is specified as zero, LIB\$GETSYI uses **node-name** and writes into the **cluster-system-id** argument the CSID corresponding to the node identified by **node-name**.

The **cluster-system-id** of a VAX node is assigned by the cluster-connection software and may be obtained by the DCL command SHOW CLUSTER. The value of the **cluster-system-id** for a VAX node is not permanent; a new value is assigned to a VAX node whenever it joins or rejoins the VAXcluster.

If **cluster-system-id** is specified as -1, LIB\$GETSYI assumes a wildcard operation and returns the requested information for each VAX node in the cluster, one node per call.

If **cluster-system-id** is not specified, **node-name** is used.

***node-name***

VMS usage: **char\_string**  
 type: **character string**  
 access: **read only**  
 mechanism: **by descriptor**

Name of the node for which information is to be returned. The **node-name** argument is the address of a descriptor pointing to the node name string. If **cluster-system-id** is not specified or is specified as zero, **node-name** is used. If neither **node-name** nor **cluster-system-id** is specified, the caller's node is used. See the **cluster-system-id** argument for more information.

The node name string must contain from 1 to 15 characters and must correspond exactly to the VAX node name; no trailing blanks nor abbreviations are permitted.

---

**DESCRIPTION**

LIB\$GETSYI provides the following features in addition to those provided by the \$GETSYI system service:

- Instead of a list of item descriptors, which may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.



# LIB\$GETSYI

- For items which return numeric values, LIB\$GETSYI can optionally provide a formatted string interpretation of the value.
- For string arguments, all string classes supported by the Run-Time Library are understood.
- Calls to LIB\$GETSYI are synchronous. LIB\$GETSYI calls LIB\$GET\_EF to allocate a local event flag number for synchronization.

All item codes that can be used with the \$GETSYI system service may be used as the **item-code** argument to LIB\$GETSYI. See the description of the \$GETSYI system service for a list of the defined item codes. Note that the symbolic names for these items are defined in DIGITAL-supplied symbol libraries in module \$SYIDEF (where appropriate).

## Value Formats

By using the **resultant-value** and **resultant-string** arguments to LIB\$GETSYI, you can request that the information be returned in two ways. For those items described as "string" in the table of Item Identifier Codes for the \$GETSYI service, the value is returned in **resultant-string**. For all other items—those which have numeric values—the numeric representation is returned in **resultant-value** (if specified), and an unsigned decimal integer representation is stored in **resultant-string**.

Each formatted item is written left-justified; **resultant-length**, if specified, gives the number of characters used. For those items which return blank-padded strings, such as SYI\$\_VERSION, trailing blanks are removed.

See the *VMS System Services Reference Manual* for a description of the \$GETSYI system service.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_STRTRU	String truncated. This is an alternate success return status. The fixed-length <b>resultant-string</b> argument could not contain all the characters of the returned item.
SS\$_BADPARAM	Unrecognized item code. <b>Item-code</b> was not recognized as valid by \$GETSYI.
LIB\$_INSEF	Insufficient event flags. A local event flag number could not be allocated by a call to LIB\$GET_EF.
LIB\$_INVARG	Invalid arguments. The \$GETSYI item identifier code describes the item as "string", and no <b>resultant-string</b> argument was specified.
LIB\$_INVSTRDES	Invalid string descriptor. The descriptor of the <b>resultant-string</b> argument is not a valid descriptor.
LIB\$_WRONUMARG	Wrong number of arguments. An incorrect number of arguments was passed to LIB\$GETSYI.

Any condition values returned by LIB\$SCOPY\_xxx.

Any condition values returned by the \$GETSYI system service.

---

## LIB\$GET\_COMMAND Get Line from SYS\$COMMAND

The Get Line from SYS\$COMMAND routine gets one record of ASCII text from the current controlling input device, specified by the logical name SYS\$COMMAND.

---

### FORMAT

**LIB\$GET\_COMMAND** *resultant-string* [,prompt-string]  
[,resultant-length]

---

### RETURNS

VMS usage: **cond\_value**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by value**

---

### ARGUMENTS

#### ***resultant-string***

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

String that LIB\$GET\_COMMAND gets from SYS\$COMMAND. The **resultant-string** argument is the address of a descriptor pointing to this string.

#### ***prompt-string***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Prompt message that LIB\$GET\_COMMAND displays on the controlling terminal. The **prompt-string** argument is the address of a descriptor pointing to the prompt. A valid prompt consists of text followed by no carriage-return /line-feed combination. A colon(:) and one space are optional.

#### ***resultant-length***

VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **write only**  
mechanism: **by reference**

Number of bytes written into **resultant-string** by LIB\$GET\_COMMAND, not counting padding in the case of a fixed string. The **resultant-length** argument is the address of an unsigned word containing this length. If the input string is truncated to the size specified in the **resultant-string** descriptor, **resultant-length** is set to this size. Therefore, **resultant-length** can always be used by the calling program to access a valid substring of **resultant-string**.

# LIB\$GET\_COMMAND

---

## DESCRIPTION

LIB\$GET\_COMMAND uses the RMS \$GET service (see the *VMS Record Management Services Manual*) to get one record of ASCII text from the current controlling input device, specified by SYS\$COMMAND.

When you log in, VMS creates three files as default I/O control streams for your process.

- SYS\$INPUT, your default input device
- SYS\$OUTPUT, your default output device
- SYS\$COMMAND, the device that supplies the commands to your process

These files remain open until you log out. They are the interface between your interactive input and output or your batch commands and the VMS software. Initially, all three files are equated with the terminal. However, with the DCL ASSIGN command, you can change these assignments to obtain information from a file or put information into a file. SYS\$INPUT and SYS\$COMMAND are usually identical, but the input and command streams can be different. For example, during the execution of an indirect command file from an interactive terminal, SYS\$COMMAND refers to the terminal and SYS\$INPUT refers to the command file.

LIB\$GET\_COMMAND opens file SYS\$COMMAND on the first call. The RMS internal stream identifier (ISI) is stored in the routine's static storage for subsequent calls.

If **prompt-string** is provided and if the SYS\$COMMAND device is a terminal, LIB\$GET\_COMMAND displays the prompt message. If the device is not a terminal, the **prompt-string** is ignored.

LIB\$GET\_COMMAND is used when a program needs input from some source other than the current input stream. Usually, it is used to input from the terminal rather than from an indirect command file. For example, a program may ask a question which cannot be answered by an indirect command file entry. In this case the program would call LIB\$GET\_COMMAND to get one record of ASCII text from SYS\$COMMAND, the terminal.

# LIB\$GET\_COMMAND

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. RMS completion status.
LIB\$_FATERRLIB	An internal consistency check on Run-Time Library data structures has failed. This may indicate a programming error in the Run-Time Library, or that your program may have overwritten those data structures.
LIB\$_INPSTRTRU	The input string has been truncated to the size specified in the <b>resultant-string</b> descriptor (fixed-length strings only). <b>Resultant-length</b> is also set to this size. This is an error (as opposed to LIB\$_STRTRU which is a success) because the truncation is not under program control.
LIB\$_INSVIRMEM	Insufficient virtual memory to allocate the dynamic string.
LIB\$_INVARG	Invalid arguments. The descriptor class field is not a recognized code or is zero.

Any valid RMS status code.

Any code returned by LIB\$GET\_VM or LIB\$SCOPY\_R\_DX.

# LIB\$GET\_COMMON

---

## LIB\$GET\_COMMON Get String from Common

The Get String from Common routine copies a string in the common area to the destination string. (The common area is an area of storage which remains defined across multiple image activations in a process.) The string length is taken from the first longword of the common area.

---

**FORMAT**            **LIB\$GET\_COMMON** *resultant-string* [,*resultant-length*]

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:          **write only**  
                          mechanism:       **by value**

---

**ARGUMENTS**         ***resultant-string***  
                          VMS usage: **char\_string**  
                          type:            **character string**  
                          access:          **write only**  
                          mechanism:       **by descriptor**

Destination string into which LIB\$GET\_COMMON writes the string copied from the common area. The **resultant-string** argument is the address of a descriptor pointing to the destination string.

***resultant-length***  
VMS usage: **word\_unsigned**  
type:        **word (unsigned)**  
access:      **write only**  
mechanism:   **by reference**

Number of characters written into **resultant-string** by LIB\$GET\_COMMON, not counting padding in the case of a fixed-length string. The **resultant-length** argument is the address of an unsigned word integer containing the number of characters copied. If the input string is truncated to the size specified in the **resultant-string** descriptor, **resultant-length** is set to this size. Therefore, **resultant-length** can always be used by the calling program to access a valid substring of **resultant-string**.

---

**DESCRIPTION**        LIB\$PUT\_COMMON allows a program to copy a string into the process's common storage area. This area remains defined during multiple image activations. LIB\$GET\_COMMON allows a program to copy a string from the common area into a destination string. The programs reading and writing the data in the common area must agree upon its amount and format.

The maximum number of characters that can be copied is 252. The actual number of characters copied is returned by the optional argument, **resultant-length** (if given).

# LIB\$GET\_COMMON

You can use LIB\$PUT\_COMMON and LIB\$GET\_COMMON to pass information between images run successively, such as chained images run by LIB\$RUN\_PROGRAM. Since the common area is unique to each process, do not use LIB\$GET\_COMMON and LIB\$PUT\_COMMON to share information across processes.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Successfully completed. The string was longer than the buffer and was truncated.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL.
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.

# LIB\$GET\_DATE\_FORMAT

---

## LIB\$GET\_DATE\_FORMAT Get the User's Date Input Format

The Get The User's Date Input Format routine returns information about the user's choice of a date/time input format.

---

<b>FORMAT</b>	<b>LIB\$GET_DATE_FORMAT</b> <i>format-string</i> <i>[,user-context]</i>
---------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>format-string</i></b> VMS usage: <b>char_string</b> type: <b>character string</b> access: <b>write only</b> mechanism: <b>by descriptor</b>
------------------	---

Receives the translation of LIB\$DT\_INPUT\_FORMAT. The **format-string** argument is the address of a descriptor pointing to this format string.

### ***user-context***

VMS usage: **context**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

Context variable that retains the translation context over multiple calls to this routine. The **user-context** argument is the address of an unsigned longword that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.

The **user-context** argument is optional. However, if a context cell is not passed, the routine LIB\$GET\_DATE\_FORMAT may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, DIGITAL recommends that users specify a different context cell for each calling thread.

---

<b>DESCRIPTION</b>	Depending on which method was used to specify the input formats, LIB\$GET_DATE_FORMAT either translates the logicals LIB\$DT_INPUT_FORMAT and LIB\$FORMAT_MNEMONICS, or uses the preinitialized context components LIB\$K_FORMAT_MNEMONICS and LIB\$K_INPUT_FORMAT to return the user's specified date/time input format in a "legible" form. This <b>format-string</b> can then be used as a guideline for entering date/time strings.
--------------------	---

# LIB\$GET\_DATE\_FORMAT

The string returned by LIB\$GET\_DATE\_FORMAT parallels the currently defined input format string, consisting of the format punctuation (with most whitespace compressed) and "legible" mnemonics representing the various format fields. The English (default) versions of these mnemonics are as follows:

Format Field	Legible Mnemonic (Default)
Year	YYYY <sup>1</sup>
Numeric month	MM
Alphabetic month	MONTH
Numeric day	DD
Hours (12- or 24-hour)	HH
Minutes	MM
Seconds	SS
Fractional seconds	CC <sup>1</sup>
Meridian indicator	AM/PM

<sup>1</sup>This variable-length field mnemonic has a numeric suffix representing the number of digits allowed/required in the field. For instance, YYYY4 indicates a four-digit year field.

For example, consider the following input format string:

```
$ DEFINE LIB$DT_INPUT_FORMAT -
_$ "!MAAU !DO, !Y2 !H02:!M0:!S0.!C4 !MIU"
```

If LIB\$GET\_DATE\_FORMAT were called for this format string, the format string returned would be as follows:

```
MONTH DD, YYYY2 HH:MM:SS.CC4 AM/PM
```

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_DEFFORUSE	Default format used; unable to determine desired format.
LIB\$_ENGLUSED	English used; unable to determine or use desired language.
LIB\$_ILLFORMAT	Illegal format string.
LIB\$_INVARG	Invalid argument; a required argument was not specified.
LIB\$_INVSTRDES	Invalid input string descriptor.
LIB\$_REENTRANCY	Reentrancy detected.
LIB\$_STRTRU	String truncated.
LIB\$_UNRFORCOD	Unrecognized format code.
LIB\$_WRONUMARG	Wrong number of arguments.

Any condition value returned by LIB\$GET\_VM, LIB\$SCOPY\_R\_DX, and LIB\$SFREE1\_DD.



# LIB\$GET\_EF

---

## LIB\$GET\_EF Get Event Flag

The Get Event Flag routine allocates one local event flag from a process-wide pool and returns the number of the allocated flag to the caller. If no flags are available, LIB\$GET\_EF returns an error as its function value.

---

**FORMAT**            **LIB\$GET\_EF** *event-flag-number*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENT**            ***event-flag-number***  
                          VMS usage: **ef\_number**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by reference**

Number of the local event flag that LIB\$GET\_EF allocated, or -1 if no local event flag was available. The **event-flag-number** argument is the address of a signed longword integer into which LIB\$GET\_EF writes the number of the local event flag that it allocates.

---

## DESCRIPTION

LIB\$GET\_EF and LIB\$FREE\_EF cause local event flags to be allocated and deallocated at run time, so that your routine remains independent of other routines executing in the same process.

The following table lists status of local event flags.

---

Number	Status
0	Never used by this routine and always available
1 to 23	Initially reserved; available after being freed by LIB\$FREE_EF
24 to 31	Reserved to VMS
32 to 63	Initially free

---

Local event flags numbered 32 to 63 are available to your program. These event flags allow routines to communicate and synchronize their operations.

Using LIB\$GET\_EF provides your program with an arbitrary event flag number. You can obtain a specific event flag number by calling LIB\$RESERVE\_EF, and passing as an argument the number of the specific event flag that you wish to reserve. However, reserving a specific local event flag number is not recommended. If you use a specific event flag in your routine, another routine may attempt to use the same flag, and the flag will

no longer function as expected. Therefore, you should call LIB\$GET\_EF to obtain the next arbitrary event flag and LIB\$FREE\_EF to return it to the storage pool.

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Routine successfully completed.

LIB\$\_INSEF

Insufficient event flags. There were no more event flags available for allocation.



**flags**

VMS usage: **mask\_longword**  
 type: **longword (unsigned)**  
 access: **modify**  
 mechanism: **by reference**

Value which LIB\$GET\_FOREIGN uses to control whether or not prompting is to be performed. The **flags** argument is the address of a signed longword integer containing this value. If the low bit of **flags** is zero, or if **flags** is omitted, prompting is done only if the CLI does not return a command line. If the low bit is 1, prompting is done unconditionally. If specified, **flags** is set to 1 before returning to the caller.

The primary use of **flags** is to allow a utility program to be invoked once with subcommand text on the command line, and then to repeatedly prompt for further subcommands from SYS\$INPUT. This is accomplished by calling LIB\$GET\_FOREIGN repeatedly, specifying in the call a **prompt-string** string and a **flags** variable which is initialized to zero at the beginning of the program. The first call gets the subcommand text from the command line, after which **flags** will be set to 1, causing further subcommands to be requested through prompts to SYS\$INPUT.

---

**DESCRIPTION**

LIB\$GET\_FOREIGN returns the contents of the command line that you use to activate an image. It can be used to give your program access to the qualifiers of a foreign command or to prompt for further command line text.

A foreign command is a command that you can define and then use as if it were a DCL or MCR command in order to run a program. When you use the foreign command at command level, the CLI parses the foreign command only and activates the image. It ignores any options or qualifiers that you have defined for the foreign command. Once the CLI has activated the image, the program can call LIB\$GET\_FOREIGN to obtain and parse the remainder of the command line (after the command itself) for whatever options it may contain. See the *VMS DCL Dictionary* for information on how to define a foreign command.

If no command line is available, LIB\$GET\_FOREIGN can optionally call LIB\$GET\_INPUT to prompt the user for command text. If desired, LIB\$GET\_FOREIGN can be called repetitively, returning the command line on the first call, but prompting for further text on subsequent calls.

LIB\$GET\_FOREIGN can also be used for images invoked by the RUN command, for which further text must be obtained by prompting. Such an image can also be invoked by the DCL command MCR or by the MCR Command Language Interpreter. The text following the image name will be returned to the executing image.

The action of LIB\$GET\_FOREIGN depends on the environment in which the image is activated.

- If you use a foreign command to invoke the image, you can call LIB\$GET\_FOREIGN to obtain the command qualifiers following the foreign command. You can also use LIB\$GET\_FOREIGN to prompt repeatedly for more qualifiers after the command. This technique is illustrated in the example.

# LIB\$GET\_FOREIGN

- If the image is in the SYS\$SYSTEM: directory, the image can be invoked by the DCL MCR command or by the MCR Command Language Interpreter. In this case, LIB\$GET\_FOREIGN returns the command line text following the image name.
- If the image is invoked by a DCL RUN command, LIB\$GET\_FOREIGN can be used to prompt for additional text.
- If the image is not invoked by a foreign command or MCR, or if there is no information remaining on the command line, and the user-supplied prompt is present, LIB\$GET\_INPUT is called to prompt for a command line. If the prompt is not present, LIB\$GET\_FOREIGN returns a zero length string.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_FATERRLIB	A fatal internal error was detected.
LIB\$_INPSTRTRU	The input string was truncated. <b>Resultant-stringing</b> could not contain all of the characters. <b>Resultant-length</b> reflects the truncated length.
LIB\$_INSVIRMEM	Insufficient virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor.

A condition value returned by RMS. SYS\$INPUT was prompted for command text and RMS returned an error. The most typical error will be RMS\$\_EOF, end-of-file.

---

## EXAMPLE

```
EXAMPLE: ROUTINE OPTIONS (MAIN);
%INCLUDE $STSDEF; /* Status-testing definitions */
DECLARE COMMAND_LINE CHARACTER(80) VARYING,
        PROMPT_FLAG FIXED BINARY(31) INIT(0),
        LIB$GET_FOREIGN ENTRY (CHARACTER(*) VARYING,
                                CHARACTER(*) VARYING,
                                FIXED BINARY(15),
                                FIXED BINARY(31))
        OPTIONS(VARIABLE) RETURNS (FIXED BINARY(31)),
        RMS$_EOF GLOBALREF FIXED BINARY(31) VALUE;
/* Repeat forever calling LIB$GET_FOREIGN to obtain
   subcommand text and print the text. Exit when an
   end-of-file is found. */
```

# LIB\$GET\_FOREIGN

```
DO WHILE ('1'B); /* Do while TRUE */
  STS$VALUE = LIB$GET_FOREIGN
              (COMMAND_LINE,'Input: ',,
              PROMPT_FLAG);
  IF STS$SUCCESS THEN
    PUT LIST (' Command was ',COMMAND_LINE);
  ELSE DO;
    IF STS$VALUE ^= RMS$_EOF THEN
      PUT LIST ('Error encountered');
    RETURN;
  END;
  PUT SKIP; /* Skip to next line */
END; /* End of DO WHILE loop */
END;
```

This PL/I example illustrates the use of the optional **flags** argument to permit repeated calls to LIB\$GET\_FOREIGN. The command line text is retrieved on the first pass only; after the first pass, the program prompts from SYS\$INPUT.



---

## DESCRIPTION

LIB\$GET\_INPUT uses the RMS \$GET service to get one record of ASCII text from the current controlling input device, specified by SYS\$INPUT. (For more information about the RMS \$GET service see the *VMS Record Management Services Manual*.)

When you log in, VMS creates three files as default I/O control streams for your process.

- SYS\$INPUT, your default input device
- SYS\$OUTPUT, your default output device
- SYS\$COMMAND, the device that supplies the commands to your process

These files remain open until you log out. They are the interface between your interactive input and output or your batch commands and the VMS software. Initially, all three names are equated with the terminal. However, with the DCL ASSIGN command, you can change these assignments to obtain information from a file or put information into a file. SYS\$INPUT and SYS\$COMMAND are usually identical, but the input and command streams can be different. For example, during the execution of an indirect command file from an interactive terminal, SYS\$COMMAND refers to the terminal and SYS\$INPUT refers to the command file.

LIB\$GET\_INPUT opens file SYS\$INPUT on the first call. The RMS internal stream identifier (ISI) is stored in the routine's static storage for subsequent calls. Hence, this routine is not AST reentrant.

If **prompt-string** is provided and the SYS\$INPUT device is a terminal, LIB\$GET\_INPUT displays the prompt message. If the device is not a terminal, the **prompt-string** argument is ignored.

If you wish to get input from some source other than the current input stream, use LIB\$GET\_COMMAND.



# LIB\$GET\_INPUT

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. RMS completion status.
LIB\$_FATERRLIB	An internal consistency check on Run-Time Library data structures has failed. This may indicate a programming error in the Run-Time Library, or that your program may have overwritten those data structures.
LIB\$_INPSTRTRU	The input string has been truncated to the size specified in the <b>resultant-string</b> descriptor (fixed-length strings only). <b>Resultant-length</b> is also set to this size. This is an error (as opposed to LIB\$_STRTRU which is a success) because the truncation is not under program control.
LIB\$_INSVIRMEM	Insufficient virtual memory to allocate the dynamic string.
LIB\$_INVARG	Invalid arguments. The descriptor class field is not a recognized code or is zero.

Any RMS condition value returned by \$GET.

Any condition value returned by LIB\$GET\_VM or LIB\$SCOPY\_R\_DX.

---

## LIB\$GET\_LUN Get Logical Unit Number

The Get Logical Unit Number routine allocates one logical unit number from a process-wide pool. If a unit is available, its number is returned to the caller. Otherwise, an error is returned as the function value.

---

**FORMAT**                    **LIB\$GET\_LUN**    *logical-unit-number*

---

**RETURNS**                    VMS usage: **cond\_value**  
                                   type:           **longword (unsigned)**  
                                   access:       **write only**  
                                   mechanism:   **by value**

---

**ARGUMENTS**                ***logical-unit-number***  
                                   VMS usage: **longword\_signed**  
                                   type:       **longword integer (signed)**  
                                   access:     **write only**  
                                   mechanism: **by reference**

Allocated logical unit number or -1 if none was available. The **logical-unit-number** argument is the address of a longword into which LIB\$GET\_LUN returns the value of the allocated logical unit. The logical unit numbers that LIB\$GET\_LUN can allocate are in the range 100 through 119.

---

**DESCRIPTION**              LIB\$GET\_LUN allocates one logical unit number from a process-wide pool. If a unit is available, its number is returned to the caller. Otherwise, an error is returned as the function value.

LIB\$GET\_LUN reserves the logical unit numbers 100 through 119. This routine assumes that the logical unit numbers in the range 0 through 99 may be in use by your program, but it cannot determine which logical unit numbers are actually in use by your program.

You should call LIB\$GET\_LUN only from FORTRAN or BASIC programs. Those languages and LIB\$GET\_LUN share the concept of unit numbers and a similar number space.

---

<b>CONDITION VALUES RETURNED</b>	SS\$_NORMAL	Routine successfully completed.
	LIB\$_INSLUN	Insufficient logical unit numbers. No logical unit numbers were available.



# LIB\$GET\_MAXIMUM\_DATE\_LENGTH

that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.

The **user-context** parameter is optional. However, if a context cell is not passed, the routine LIB\$GET\_MAXIMUM\_DATE\_LENGTH may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, DIGITAL recommends that users specify a different context cell for each calling thread.

## **flags**

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Bit mask that allows the user to specify whether the date, time, or both are to be included in the calculation of the maximum date length. The **flags** argument is the address of an unsigned bit mask containing the specified values. Valid values are LIB\$M\_DATE\_FIELDS and LIB\$M\_TIME\_FIELDS. The values specified for **flags** must correspond to the **flags** argument passed to LIB\$FORMAT\_DATE\_TIME.

---

## DESCRIPTION

The LIB\$GET\_MAXIMUM\_DATE\_LENGTH routine determines the maximum possible length for a formatted date/time string as returned by LIB\$FORMAT\_DATE\_TIME. The maximum length returned takes into account the currently specified output format and natural language; **date-length** represents the maximum possible length of the string written to the **date-string** argument of LIB\$FORMAT\_DATE\_TIME.

Consider the following example, in which the output format is defined as follows.

```
DEFINE LIB$DT_FORMAT LIB$DATE_FORMAT_012, LIB$TIME_FORMAT_012
```

This date/time format would appear as follows:

```
!MAU !DD, !Y4 !HH2:!MO !MIU
```

Given this format, the maximum possible length for this date/time string is calculated using the longest possible date string followed by a space and the longest possible time string. One example that meets these requirements is as follows (assuming English as the selected language):

```
SEPTEMBER 21, 1988 11:24 PM
```

The maximum possible length of this **date-string** would then be 28.

# LIB\$GET\_MAXIMUM\_DATE\_LENGTH

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_ENGLUSED	English used by default; unable to translate SYS\$LANGUAGE.
LIB\$_DEFFORUSE	Default format used; unable to determine desired format.
LIB\$_UNRFORCOD	Unrecognized format code.
LIB\$_STRTRU	Output string truncated.
LIB\$_ABSTIMREQ	Absolute time required.
LIB\$_REENTRANCY	Reentrant invocation with same context variable.

Any condition value returned by LIB\$GET\_VM.



# LIB\$GET\_SYMBOL

## ***table-type-indicator***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **write only**  
mechanism: **by reference**

Indicator of which table contained the symbol. The **table-type-indicator** argument is the address of a signed longword integer into which LIB\$GET\_SYMBOL writes the table indicator.

Possible values of the table indicator are listed below.

Symbolic Name	Value	Table
LIB\$K_CLI_LOCAL_SYM	1	Local symbol table
LIB\$K_CLI_GLOBAL_SYM	2	Global symbol table

LIB\$K\_CLI\_LOCAL\_SYM and LIB\$K\_CLI\_GLOBAL\_SYM are defined in DIGITAL-supplied symbol libraries (macro or module name \$LIBCLIDEF) and as global symbols.

---

## **DESCRIPTION**

LIB\$GET\_SYMBOL first searches the local symbol table for the symbol name, then searches the global symbol table. Numeric values are converted to an ASCII representation of a signed decimal number before being returned.

LIB\$GET\_SYMBOL is supported for use with the DCL Command Language Interpreter. If used with the MCR CLI, the error status LIB\$\_NOCLI will be returned.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to get the symbol. In that case, LIB\$GET\_SYMBOL returns the error status LIB\$\_NOCLI.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Routine successfully completed; string truncated. The destination string could not contain all the characters in the symbol value.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSCLIMEM	Insufficient CLI memory. The CLI could not obtain enough virtual memory to perform the function. This may be caused by having too many symbols defined. Deleting some symbol definitions may relieve the situation.
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.
LIB\$_INVSYMNAM	Invalid symbol name. The symbol name contained more than 255 characters or did not begin with a letter or dollar sign (\$).
LIB\$_NOCLI	No CLI present. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_NOSUCHSYM	No such symbol. The symbol was not defined in either the local or global symbol table.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL Command Language Interpreter, please report the problem to DIGITAL in a Software Performance Report (SPR).



# LIB\$GET\_USERS\_LANGUAGE

---

## LIB\$GET\_USERS\_LANGUAGE Return the User's Language

The Return the User's Language routine determines the user's choice of a natural language. The choice is determined by translating the logical SYS\$LANGUAGE.

---

**FORMAT**            **LIB\$GET\_USERS\_LANGUAGE**    *language*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:         **write only**  
                          mechanism:     **by value**

---

**ARGUMENTS**         ***language***  
                          VMS usage: **char\_string**  
                          type:           **character string**  
                          access:         **write only**  
                          mechanism:     **by descriptor**

Receives the translation of SYS\$LANGUAGE. The **language** argument is the address of a descriptor pointing to this language name.

---

**DESCRIPTION**        The LIB\$GET\_USERS\_LANGUAGE routine translates the logical SYS\$LANGUAGE and returns the user's choice of a natural language. If the logical SYS\$LANGUAGE does not translate for some reason, then the language defaults to English and the status LIB\$\_ENGLUSED is returned.

If a failure or truncation occurs while copying the language name to the **language** string argument, that error status overrides the LIB\$\_ENGLUSED or SS\$\_NORMAL status.

---

**CONDITION VALUES RETURNED**

SS\$_NORMAL	Normal successful completion.
LIB\$_ENGLUSED	English used by default; unable to translate SYS\$LANGUAGE.

Any condition value returned by LIB\$SCOPY\_R\_DX.

---

## LIB\$GET\_VM Allocate Virtual Memory

The Allocate Virtual Memory routine allocates a specified number of contiguous bytes in the program region and returns the virtual address of the first byte allocated.

---

**FORMAT**            **LIB\$GET\_VM** *number-of-bytes, base-address [,zone-id]*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:        **longword (unsigned)**  
                           access:     **write only**  
                           mechanism: **by value**

---

**ARGUMENTS**        ***number-of-bytes***  
                           VMS usage: **longword\_signed**  
                           type:        **longword integer (signed)**  
                           access:     **read only**  
                           mechanism: **by reference**

Number of contiguous bytes that LIB\$GET\_VM allocates. The **number-of-bytes** argument is the address of a longword integer containing the number of bytes. LIB\$GET\_VM allocates enough memory to satisfy the request. Your program should not reference an address before the first byte address allocated (**base-address**) or beyond the last byte allocated (**base-address + number-of-bytes - 1**) since that space may be assigned to another routine. The value of **number-of-bytes** must be greater than zero.

### ***base-address***

VMS usage: **address**  
 type:        **longword (unsigned)**  
 access:     **write only**  
 mechanism: **by reference**

First virtual address of the contiguous block of bytes allocated by LIB\$GET\_VM. The **base-address** argument is the address of an unsigned longword containing this base address.

### ***zone-id***

VMS usage: **identifier**  
 type:        **longword (unsigned)**  
 access:     **read only**  
 mechanism: **by reference**

The **zone-id** argument is the address of a longword that contains a zone identifier created by a previous call to LIB\$CREATE\_VM\_ZONE or LIB\$CREATE\_USER\_VM\_ZONE. This argument is optional. If **zone-id** is omitted or if the longword contains the value 0, LIB\$VM's default zone is used.

# LIB\$GET\_VM

---

## DESCRIPTION

LIB\$GET\_VM satisfies an allocation request by reusing free memory in the zone, or by obtaining additional memory from the processwide page pool managed by LIB\$GET\_VM\_PAGE.

LIB\$GET\_VM rounds up the value of **number-of-bytes** to a multiple of the **block-size** specified for the zone. The first byte allocated is aligned on the boundary specified by the alignment value for the zone.

If you specified allocation filling when you created the zone, LIB\$GET\_VM will fill each byte allocated. Otherwise, LIB\$GET\_VM does not initialize the memory and its contents are unpredictable.

All memory allocated by LIB\$GET\_VM has user mode read/write access, even if the call to LIB\$GET\_VM was made from a more privileged access mode.

The space allocated by successive calls to LIB\$GET\_VM may be noncontiguous because another routine can call LIB\$GET\_VM between your calls. If AST interrupts occur, LIB\$GET\_VM may allocate space to another routine between execution of any two statements in your program. Even if successive calls to LIB\$GET\_VM return two contiguous blocks, you must not combine them into one large block that is freed by a single call to LIB\$FREE\_VM.

LIB\$GET\_VM is fully reentrant, so it may be called by routines executing at AST level or in an Ada multitasking environment.

Your program must retain the address of the allocated area. This allows you to access or deallocate the space later.

If the zone you are getting was created using the LIB\$CREATE\_USER\_VM\_ZONE routine, then you must have an appropriate action routine for the get operation. That is, in your call to LIB\$CREATE\_USER\_VM\_ZONE, you must have specified a **user-get-routine**.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_INSVIRMEM	Insufficient virtual memory. The request required more dynamic memory than was available from the operating system. No partial allocation is made in this case.
LIB\$_BADBLOSIZ	Bad block size. The value of <b>number-of-bytes</b> was less than or equal to zero. For the fixed size blocks algorithm, LIB\$_BADBLOSIZ can also be generated if the value of <b>algorithm-argument</b> specified in the call to LIB\$CREATE_VM_ZONE is less than <b>number-of-bytes</b> .
LIB\$_BADBLOADR	Invalid <b>zone-id</b> .
LIB\$_PAGLIMEXC	Allocation exceeds the zone's <b>page-limit</b> .

---

## LIB\$GET\_VM\_PAGE Get Virtual Memory Page

The Get Virtual Memory Page routine allocates a specified number of contiguous pages of memory in the program region and returns the virtual address of the first page allocated.

---

**FORMAT**            **LIB\$GET\_VM\_PAGE** *number-of-pages ,base-address*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:       **longword (unsigned)**  
                           access:     **write only**  
                           mechanism: **by value**

---

**ARGUMENTS**        ***number-of-pages***  
                           VMS usage: **longword\_signed**  
                           type:       **longword integer (signed)**  
                           access:     **read only**  
                           mechanism: **by reference**

Number of pages. The **number-of-pages** argument is the address of a longword integer which specifies the number of contiguous pages to be allocated. The value of **number-of-pages** must be greater than zero.

***base-address***

VMS usage: **address**  
 type:       **longword (unsigned)**  
 access:     **write only**  
 mechanism: **by reference**

Block address. The **base-address** argument is the address of a longword which is set to the address of the first byte of the newly allocated block of pages.

---

**DESCRIPTION**        LIB\$GET\_VM\_PAGE allocates blocks of contiguous (512 byte) pages in the program region. LIB\$GET\_VM\_PAGE manages a processwide pool of free pages. If there are not enough contiguous free pages to satisfy an allocation request, additional pages are created by calling the system service \$EXPREG. All memory allocated by LIB\$GET\_VM\_PAGE is page aligned; that is, the low-order nine bits of the base address are zero.

All memory allocated by LIB\$GET\_VM\_PAGE has user-mode read/write access, even if the call to LIB\$GET\_VM\_PAGE is made from a more privileged access mode.

The contents of memory allocated by LIB\$GET\_VM\_PAGE are unpredictable. Your program must assign values to all locations that it uses.

LIB\$GET\_VM\_PAGE is designed for request sizes ranging from one page to a few hundred pages. For very large request sizes (over 1000 pages in a single request), you should call the system service \$EXPREG.

# LIB\$GET\_VM\_PAGE

LIB\$GET\_VM\_PAGE is fully reentrant, so it can be called by routines executing at AST level or in an Ada multitasking environment.

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Normal successful completion.

LIB\$\_BADBLOSIZ

The value of the argument **num\_pages** is less than or equal to 0.

LIB\$\_INSVIRMEM

Insufficient virtual memory. The request required more dynamic memory than was available from the operating system. No partial allocation is made in this case.

---

## LIB\$ICHAR Convert First Character of String to Integer

The Convert First Character of String to Integer routine converts the first character of a source string to an 8-bit ASCII integer extended to a longword.

---

**FORMAT**            **LIB\$ICHAR** *source-string*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                          type:            **longword (unsigned)**  
                          access:        **write only**  
                          mechanism:    **by value**

First character of the source string. This character is returned by LIB\$ICHAR as an 8-bit ASCII value extended to a longword. If the source string has zero length, LIB\$ICHAR returns a zero.

---

**ARGUMENT**            ***source-string***  
                          VMS usage: **char\_string**  
                          type:            **character string**  
                          access:        **read only**  
                          mechanism:    **by descriptor**

Source string whose first character is converted to an integer by LIB\$ICHAR. The **scr-str** argument is the address of a descriptor pointing to this source string.

---

**DESCRIPTION**        Although FORTRAN users can call LIB\$ICHAR, it is more efficient to use the FORTRAN intrinsic function ICHAR, which generates equivalent code in line.

---

**CONDITION**            *None.*  
**VALUES**  
**RETURNED**

# LIB\$ICHAR

---

## EXAMPLE

```
PROGRAM ICHAR(INPUT, OUTPUT);
{+}
{ This program demonstrates how to call LIB$ICHAR
{ to convert the first character of string to an
{ integer value.
{-}

FUNCTION LIB$ICHAR(SRCSTR : VARYING [A] OF CHAR) : INTEGER;
    EXTERN;

{+}
{ Declare the variables to be used.
{-}

VAR
    CHARSTR      : VARYING [256] OF CHAR;
    RET_STATUS   : INTEGER;

{+}
{ Begin the main program. Read the character string.
{ call LIBN$ICHAR, and print the result.
{-}

BEGIN
    WRITELN('Enter string: ');
    READLN(CHARSTR);
    RET_STATUS := LIB$ICHAR(CHARSTR);
    WRITELN(RET_STATUS);
END.
```

The output generated by this Pascal program is as follows:

```
$ RUN ICHAR
Enter string:
Pencil sharpener
      80
$ RUN ICHAR
Enter string:
pencil sharpener
      112
```

Notice that this routine changes any uppercase characters to lowercase.

---

## LIB\$INDEX Index to Relative Position of Substring

The Index to Relative Position of Substring routine returns an index, which is the relative position of the first occurrence of a substring in the source string.

---

**FORMAT**            **LIB\$INDEX** *source-string ,sub-string*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                          type:            **longword (unsigned)**  
                          access:         **write only**  
                          mechanism:     **by value**

The relative position of the first character of the substring if found, or zero if not found.

---

**ARGUMENTS**         ***source-string***  
                          VMS usage: **char\_string**  
                          type:            **character string**  
                          access:         **read only**  
                          mechanism:     **by descriptor**

Source string to be searched by LIB\$INDEX. The **source-string** argument is the address of a descriptor pointing to this source string.

***sub-string***  
VMS usage: **char\_string**  
type:            **character string**  
access:         **read only**  
mechanism:     **by descriptor**

Substring to be found. The **sub-string** argument is the address of a descriptor pointing to this substring.

---

**DESCRIPTION**        The relative character positions returned by LIB\$INDEX are numbered 1, 2, ..., n. Thus, zero means that the substring was not found.

If the substring has a zero length, LIB\$INDEX returns the value 1, indicating success, no matter how long the source string is. If the source string has a zero length and the substring has a nonzero length, zero is returned, indicating that the substring was not found.

FORTRAN users may use the built-in INDEX function rather than calling LIB\$INDEX directly.



# LIB\$INDEX

---

**CONDITION  
VALUES  
RETURNED**

*None.*



# LIB\$INIT\_DATE\_TIME\_CONTEXT

- LIB\$K\_MONTH\_NAME
- LIB\$K\_MONTH\_NAME\_ABB
- LIB\$K\_FORMAT\_MNEMONICS
- LIB\$K\_WEEKDAY\_NAME
- LIB\$K\_WEEKDAY\_NAME\_ABB
- LIB\$K\_RELATIVE\_DAY\_NAME
- LIB\$K\_MERIDIAN\_INDICATOR
- LIB\$K\_OUTPUT\_FORMAT
- LIB\$K\_INPUT\_FORMAT

## *init-string*

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

The characters which are to be used in formatting dates and times for input or output. The **init-string** argument is the address of a descriptor pointing to this string.

---

## DESCRIPTION

The LIB\$INIT\_DATE\_TIME\_CONTEXT routine allows the user to initialize the context area used by either LIB\$CONVERT\_DATE\_STRING or LIB\$FORMAT\_DATE\_TIME with specific strings instead of through logical name translations. This routine is therefore useful when the application is formatting either input or output strings that are used only by other computer applications and are not intended for presentation to users.

When an application initializes the context area using this routine, it is expected that all required context information will be provided in this way. In other words, it is not expected that some items will be initialized and other items will be gathered through logical name translation.

Therefore, the minimum effort required to initialize the necessary format strings would be a call to LIB\$INIT\_DATE\_TIME\_CONTEXT specifying the input or output format strings to be used. If the specified format requires spelled items, such as month names or day names, then additional calls to LIB\$INIT\_DATE\_TIME\_CONTEXT are required to provide the spellings of these items.

The format of the strings used by this routine is as follows:

"[**delim**][string-1][**delim**][string-2][**delim**] . . . [**delim**][string-n][**delim**]"

In this format, [**delim**] is any character that is not in any of the strings, and [string-x] is the spelling of that instance of the component.

For example, a string passed to this routine to specify the English spellings of the month names might be as follows:

"|JAN|FEB|MAR|APR|MAY|JUN |JUL|AUG|SEP|OCT|NOV|DEC|"

# LIB\$INIT\_DATE\_TIME\_CONTEXT

Note that the string starts and ends with a delimiter. Thus, there is one more delimiter than there are string elements. Each type of component has a natural number of elements associated. The string must contain exactly that number of elements.

Month names (full or abbreviated)	12
Format mnemonics	9
Day names (full or abbreviated)	7
Relative day names	3
Meridian indicators	2
Output format strings	2
Input format string	1

In order to specify the input format mnemonics using LIB\$INIT\_DATE\_TIME\_CONTEXT, the user must initialize the component LIB\$K\_FORMAT\_MNEMONICS with the appropriate values. The following table lists in order the 9 fields that must be initialized, along with their default (English) values.

Order	Format Field	Legible Mnemonic (Defaults)
1	Year	YYYY
2	Numeric month	MM
3	Numeric day	DD
4	Hours (12- or 24-hour)	HH
5	Minutes	MM
6	Seconds	SS
7	Fractional seconds	CC
8	Meridian indicator	AM/PM
9	Alphabetic month	MONTH

For example, the following would be a valid definition of LIB\$K\_FORMAT\_MNEMONICS using Austrian as the natural language:

```
|JJJJ|MM|TT|SS|MM|SS|HH| |MONAT|
```

To specify an output format using LIB\$INIT\_DATE\_TIME\_CONTEXT, the user must initialize the variable LIB\$K\_OUTPUT\_FORMAT. There are two elements associated with this output format string. One describes the date format fields, the other the time format fields. The order in which they appear in the string determines the order in which they are output. A single space is inserted into the output stream between the two elements, if the call to LIB\$FORMAT\_DATE\_TIME specifies that both be output. For example:

```
"|!DB-!MAAU-!Y4|!H04:!M0:!S0.!C2|"
```

This output format string represents the format used by the \$ASCTIM system service for outputting times. Note that the middle delimiter is replaced by a space in the resultant output.

A more detailed description of the format mnemonics used in these routines is given in the introductory section of this manual.

# LIB\$INIT\_DATE\_TIME\_CONTEXT

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Normal successful completion.
LIB\$_NUMELEMENTS	Incorrect number of elements for the component.
LIB\$_ILLINISTR	Illegally formed <b>init-string</b> .
LIB\$_UNRFORCOD	Unrecognized format code.
LIB\$_ILLCOMPONENT	Illegal value for the component.

Any condition value returned by LIB\$GET\_VM.

Any condition value returned by LIB\$ANALYZE\_SDESC.

---

## LIB\$INIT\_TIMER Initialize Times and Counts

The Initialize Times and Counts routine stores the current values of specified times and counts for use by LIB\$SHOW\_TIMER or LIB\$STAT\_TIMER.

---

**FORMAT**            **LIB\$INIT\_TIMER** [*context*]

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENT**            **context**  
                           VMS usage: **context**  
                           type:            **longword (unsigned)**  
                           access:        **modify**  
                           mechanism:    **by reference**

Context variable that retains the values of the times and counts. The **context** argument contains the address of an unsigned longword that is this context. When you call LIB\$INIT\_TIMER, you must use the optional **context** argument only if you want to maintain several sets of statistics simultaneously.

- If **context** is omitted, the control block is allocated in static storage. This method is not AST reentrant.
- If **context** is zero, a control block is allocated in dynamic heap storage. The times and counts will be stored in that block and the address of the block returned in **context**. This method is fully reentrant and modular.
- If **context** is nonzero, it is considered to be the address of a control block previously allocated by a call to LIB\$INIT\_TIMER. If so, the control block is reused, and fresh times and counts are stored in it.

When LIB\$INIT\_TIMER returns, the block of storage referred to by **context** will contain the times and counts.

---

**DESCRIPTION**        LIB\$INIT\_TIMER stores the current values of specified times and counts in one of three places, depending on the value of the optional **context** argument. You need to call LIB\$FREE\_TIMER only if you have specified **context** in LIB\$INIT\_TIMER and you wish to deallocate all heap storage resources.

# LIB\$INIT\_TIMER

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Routine successfully completed.

LIB\$\_INVARG

Invalid argument; **context** is nonzero and the block to which it refers was not initialized on a previous call to LIB\$INIT\_TIMER.

LIB\$\_INSVIRMEM

**Context** is zero, and there is insufficient virtual memory to allocate a storage block.





# LIB\$INSERT\_TREE

## ***user-compare-routine***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied compare routine that LIB\$INSERT\_TREE calls to compare a symbol with a node. The **user-compare-routine** argument is the address of the entry mask to the compare routine. The **user-compare-routine** argument is required; LIB\$INSERT\_TREE calls the compare routine for every node except the first node in the tree. The value returned by the compare routine indicates the relationship between the symbol key and the node.

For more information on the compare routine, see "Call Format for a Compare Routine" in the Description section below.

## ***user-allocation-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied allocate routine that LIB\$INSERT\_TREE calls to allocate virtual memory for a node. The **user-allocation-procedure** argument is the address of the entry mask to the allocate routine. The **user-allocation-procedure** argument is required; LIB\$INSERT\_TREE always calls the allocate routine.

For more information on the allocate routine, see "Call Format for an Allocate Routine" in the Description section below.

## ***new-node***

VMS usage: **address**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by reference**

Location where the new key is inserted. The **new-node** argument is the address of an unsigned longword that is the address of the new node.

## ***user-data***

VMS usage: **user\_arg**  
type: **unspecified**  
access: **unspecified**  
mechanism: **by value**

User data that LIB\$INSERT\_TREE passes to the compare and allocate routines. **User-data** is an optional argument.

---

## **DESCRIPTION**

This Description section contains three parts.

- Guidelines for using LIB\$INSERT\_TREE
- Call format for a compare routine
- Call format for an allocate routine

# LIB\$INSERT\_TREE

## Guidelines for Using LIB\$INSERT\_TREE

LIB\$INSERT\_TREE inserts a node in a balanced binary tree. You supply two routines: compare and allocate. The compare routine compares the symbol key to the node, and the allocate routine allocates virtual memory for the node to be inserted. LIB\$INSERT\_TREE first calls the compare routine to find the location at which the new node is inserted. Then LIB\$INSERT\_TREE calls the allocate routine to allocate memory for the new node. Most programmers insert data in the new node by initializing it within the allocate routine as soon as memory has been allocated.

You may pass the data to be inserted into the tree using either the **symbol** argument alone or both the **symbol** and **user-data** arguments. The **symbol** argument is required. It may contain all of the data, just the name of the node, or the address of the data. If you decide to use **symbol** to pass just the name of the node, you must use the **user-arg** argument to pass the rest of the data to be inserted in the new node.

## Call Format for a Compare Routine

The call format of a compare routine is as follows:

```
user-compare-routine  symbol ,comparison-node [,user-data]
```

LIB\$INSERT\_TREE passes both the **symbol** and **comparison-node** arguments to the compare routine, using the same passing mechanism that was used to pass them to LIB\$INSERT\_TREE. The **user-data** argument is passed in the same way, but its use is optional.

The **user-compare-routine** argument in the call to LIB\$INSERT\_TREE specifies the compare routine. This argument is required. LIB\$INSERT\_TREE calls the compare routine for every node except the first node in the tree.

The value returned by the compare routine is the result of comparing the symbol key with the current node. Listed below are the possible values returned by the compare routine.

Value	Description
Negative	<b>symbol</b> is less than the current node
Zero	<b>symbol</b> is equal to the current node
Positive	<b>symbol</b> is greater than the current node

This is an example of a user-supplied compare routine written in BASIC.

```
FUNCTION LONG Compare_node (                                     &
                                STRING  Key_string,             &
                                Node_type Node,                 &
                                LONG    Dummy)
!+
!   This function compares the string described by Key_string with
!   the string contained in the data node Node, and returns 0
!   if the strings are equal, -1 if Key_string is < Node, and
!   1 if Key_string > Node.
!-

OPTION TYPE = EXPLICIT
```

# LIB\$INSERT\_TREE

```
RECORD Node_type
  BYTE      Header (9)          ! Header
  BYTE      Length             ! Length
  STRING    Text = 80          ! String
END RECORD Node_type

DECLARE STRING Node_string

!+
! Return the result of the comparison.
!-

Node_string = SEG$ (Node::Text, 1, Node::Length)

SELECT Key_string
CASE < Node_string
  Compare_node = -1%
CASE > Node_string
  Compare_node = 1%
CASE ELSE
  Compare_node = 0%
END SELECT

END FUNCTION
```

## Call Format for an Allocate Routine

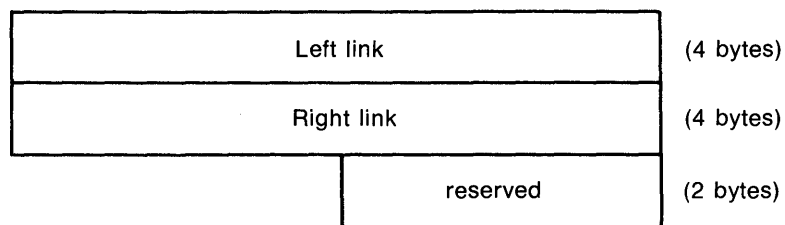
LIB\$INSERT\_TREE calls the allocate routine to allocate virtual memory for a node. The allocate routine then stores the value of **user-data** in the field of the allocated node.

The format of the call is as follows:

**user-allocation-procedure** symbol ,new-node [,user-data]

LIB\$INSERT\_TREE passes the **symbol**, **new-node**, and **user-data** arguments to your allocate routine, using the same passing mechanisms that were used to pass them to LIB\$INSERT\_TREE. Use of user data is optional.

A node header appears at the beginning of each node. The following figure illustrates the structure of a node header.



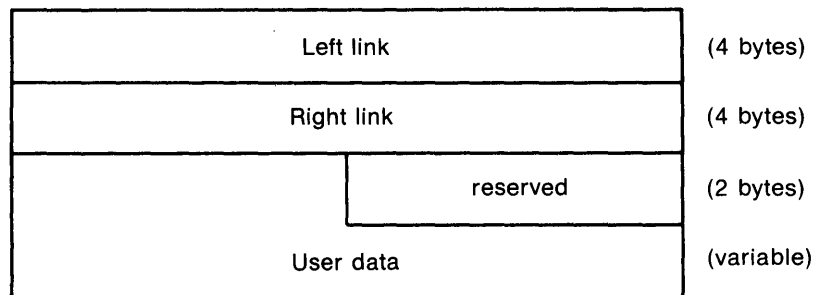
ZK-1926-84

Therefore, any node you declare that LIB\$INSERT\_TREE manipulates must contain 10 bytes of reserved data at the beginning for the node header.

How a node is structured depends on how you allocate your user data. You can allocate data in one of two ways:

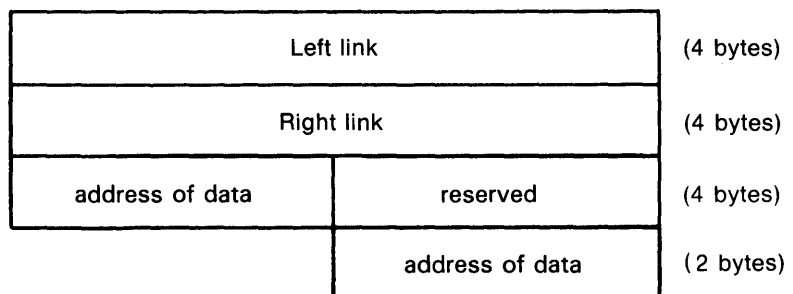
- 1 Your data immediately follows the node header. In this case your allocation routine must allocate a block equal in size to the sum of your data plus 10 bytes for the node header.

# LIB\$INSERT\_TREE



ZK-1927-84

- 2 The node contains the 10 bytes of header information and a longword pointer to the user data.



ZK-1928-84

The **user-allocation-procedure** argument in the call to LIB\$INSERT\_TREE specifies the allocate routine. This argument is required. LIB\$INSERT\_TREE always calls the allocate routine.

This is an example of a user-supplied allocate routine written in BASIC.

```

FUNCTION LONG Alloc_node (                                     &
                                STRING Key_string,           &
                                LONG Ret_addr,               &
                                LONG Dummy)
!+
! Allocate virtual memory for a new node. KEY_STRING
! is a descriptor of the string to enter into the newly
! allocated node. RET_ADDR will contain the address
! of the allocated memory.
!-

OPTION TYPE = EXPLICIT
DECLARE LONG Status_code
EXTERNAL LONG FUNCTION LIB$GET_VM
EXTERNAL SUB LIB$MOVC3
!+
! Allocate node: 10 for header, 1 for length plus length of string
!-
Status_code = LIB$GET_VM (10% + LEN (Key_string) + 1%, Ret_addr)
CALL LIB$STOP (Status_code) IF (Status_code AND 1%) <> 1%

```

# LIB\$INSERT\_TREE

```
!+
! Store key string length in byte 11 of node. Note that LIB$MOVC3
! accepts its arguments by reference.
!-
      CALL LIB$MOVC3 (1%, LEN (Key_string), (Ret_addr + 10%) BY VALUE)
!+
! Store key string in bytes 12:n of node.
!-
      CALL LIB$MOVC3 (LEN (Key_string), Key_string BY REF, &
                    (Ret_addr + 11%) BY VALUE)

      Alloc_node = 1%
END FUNCTION
```

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Routine successfully completed.
LIB\$_KEYALRINS	Routine successfully completed, but a key was found in the tree. A new key was not inserted.
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program's virtual memory requirements have exceeded either your process quota (PGFLQUOTA) or the system parameter VIRTUALPAGCNT.

---

## EXAMPLE

```
1 %TITLE 'LIB$ Tree Example In BASIC V2'
  %SBTTL 'Main Program'

!+
! This program will ask the user to enter a series of strings,
! one per line. The user will then be permitted to query the
! program to find strings that were previously entered. At the
! end, the entire tree will be displayed, along with the sequence
! number that indicates the order in which the element was entered.
!
! This program should serve as an example of the use of LIB$INSERT_TREE,
! LIB$LOOKUP_TREE and LIB$TRAVERSE_TREE.
!-

OPTION TYPE = EXPLICIT          ! Everything must be declared
DECLARE INTEGER CONSTANT True = -1 ! Useful named constant
COMMON STRING Text_line = 80    ! Allocate static line buffer
DECLARE LONG                    ! Some variables          &
      Tree_head                 ! Head for the tree      &
      ,New_node                 ! New node after insert &
      ,Status_code              ! Return status code    &
      ,Counter                  ! Sequence number
RECORD Tree_element             ! Define the structure of our
      LONG Seq_num              ! record. This record could
      LONG Text_len             ! contain many useful data items
      STRING Text = 80
END RECORD Tree_element
```

# LIB\$INSERT\_TREE

```

DECLARE Tree_element Rec          ! Declare an instance of the record
EXTERNAL LONG FUNCTION LIB$INSERT_TREE ! Function to insert node
EXTERNAL SUB LIB$TRAVERSE_TREE      ! Routine to walk tree
EXTERNAL LONG FUNCTION LIB$LOOKUP_TREE ! Routine to find a node
EXTERNAL SUB LIB$STOP               ! Routine to signal fatal error
EXTERNAL LONG                      ! Routine entry points      &
    Compare_node_2                 ! Compare entry (2 arg)  &
    ,Compare_node_3                ! Compare entry (3 arg)  &
    ,Alloc_node                    ! Allocation entry      &
    ,Print_node                    ! Print entry for walk
EXTERNAL LONG CONSTANT LIB$_KEYNOTFOU ! Key not found

!+
! Initialize the tree to null and open the terminal for I/O.
!-
Tree_head = 0%
OPEN 'SYS$INPUT' AS FILE #1
PRINT 'Enter one word per line, ^Z to begin searching the tree'
ON ERROR GOTO Input_EOF          ! Establish error handler

10 !+
! Loop, reading lines of text until the end of the file.
!-

Counter = 0
WHILE True
    LINPUT #1, '> '; Text_line
    Counter = Counter + 1
    Rec::Seq_Num = Counter
    Rec::Text_len = LEN(TRM$(Text_line))
    Rec::Text = TRM$(Text_line)
    Status_code = LIB$INSERT_TREE ( ! Insert entry into the tree      &
                                    Tree_head, Rec, 1%,              &
                                    Compare_node_3, Alloc_node, New_node)
    CALL LIB$STOP (Status_code BY VALUE) IF (Status_code AND 1%) <> 1%
NEXT

20 !+
! End of file encountered. Begin searching the tree.
!-

PRINT
PRINT "You will now be prompted for words to find. Enter one per line."
Rec::Seq_num = -1%
WHILE True
    PRINT
    LINPUT #1, 'Word to find? '; Text_line
    Rec::Text_len = LEN(TRM$(Text_line))
    Rec::Text = TRM$(Text_line)
    Status_code = LIB$LOOKUP_TREE (Tree_head, Rec, Compare_Node_2, New_node)
    IF Status_code = LIB$_KEYNOTFOU
    THEN
        PRINT "The word you entered does not appear in the tree"
    ELSE
        CALL Display_Node (New_node BY VALUE)
    END IF
NEXT

```

# LIB\$INSERT\_TREE

30

```
!+
! The user has finished searching the tree for specific items. It
! is now time to traverse the entire tree.
!-

PRINT
PRINT "The following is a dump of the tree. Notice that the words"
PRINT "are in alphabetical order"

CALL LIB$TRAVERSE_TREE (Tree_head, Print_node, 0% BY VALUE)

GOTO End_of_program
```

Input\_EOF:

```
!+
! This is the handler for an exception from INPUT.
!-

IF ERR = 11                ! End of file
THEN
    SELECT ERL
        CASE 10            ! No more input, begin searching tree
            RESUME 20
        CASE 20            ! No more items to search for, traverse
            RESUME 30      ! tree
        CASE ELSE
            ON ERROR GO BACK ! Resignal the error
    END SELECT
ELSE
    ON ERROR GO BACK      ! Resignal the error
END IF
```

End\_of\_program:

```
!+
! This is the end of the program.
!-

END
```

```
%TITLE 'LIB$ Tree Example in BASIC V2'
%SBTTL 'Function to print a node during tree traversal'
100 FUNCTION LONG Print_node (Node_type Node, LONG Dummy)
```

```
!+
! Print the string contained in the current node.
!-

OPTION TYPE = EXPLICIT

RECORD Node_type
    BYTE      Header (9)      ! Header
    LONG      Seq_Num         ! Sequence number
    LONG      Length          ! Length
    STRING    Text = 80       ! String
END RECORD Node_type

PRINT Node::Seq_Num, SEG$ (Node::Text, 1%, Node::length)

Print_node = 1%
END FUNCTION
```

# LIB\$INSERT\_TREE

```

%TITLE 'LIB$ Tree Example in BASIC V2'
%SBTTL 'Function to allocate VM for a node'
200 FUNCTION LONG Alloc_node (
                                Tree_element   Rec,
                                LONG            Ret_addr,
                                LONG            Dummy)
                                &
                                &
                                &
!+
! Allocate virtual memory for a new node. Rec is the
! data record to be entered into the newly
! allocated node. RET_ADDR will contain the address
! of the allocated memory.
!-

OPTION TYPE = EXPLICIT

RECORD Tree_element
    LONG      Seq_num
    LONG      Text_len
    STRING    Text = 80
END RECORD Tree_element

! Define the structure of our
! record. This record could
! contain many useful data items

DECLARE LONG Status_code, Data_len

EXTERNAL LONG FUNCTION LIB$GET_VM

EXTERNAL SUB LIB$MOVC3

!+
! Calculate the length of our data.
!-

    Data_len = 8% + Rec::Text_len

!+
! Allocate node: 10 for header, plus the length of our data.
!-

    Status_code = LIB$GET_VM (10% + Data_len, Ret_addr)
    CALL LIB$STOP (Status_code) IF (Status_code AND 1%) <> 1%

!+
! Store the data in the newly allocated virtual memory. Note
! that we pass the first field by reference, which is the same
! as passing the address of the entire record. We add 10 to the
! address of the VM and pass the result by value so LIB$MOVC3
! receives the address that marks the beginning of our data in
! the node.
!-

    CALL LIB$MOVC3 (Data_Len, Rec::Seq_Num, (Ret_addr + 10%) BY VALUE)

    Alloc_node = 1%

END FUNCTION

%TITLE 'LIB$ Tree Example in BASIC V2'
%SBTTL 'Function to compare two nodes'
300 FUNCTION LONG Compare_node_3 (
                                Tree_element   Rec,
                                Node_type     Node,
                                LONG            Dummy)
                                &
                                &
                                &

OPTION TYPE = EXPLICIT

```



# LIB\$INSERT\_TREE

```
RECORD Tree_element          ! Define the structure of our
  LONG   Seq_num             ! record. This record could
  LONG   Text_len           ! contain many useful data items
  STRING Text = 80
END RECORD Tree_element

RECORD Node_type
  BYTE   Header (9)         ! Header
  LONG   Seq_Num            ! Sequence number
  LONG   Length             ! Length
  STRING Text = 80         ! String
END RECORD Node_type

EXTERNAL LONG FUNCTION Compare_node_2

!+
! Call the 2 argument version of the compare routine.
!-

Compare_node_3 = Compare_node_2 ( Rec, Node )

END FUNCTION

310 FUNCTION LONG Compare_node_2 (                               &
                                Tree_element   Rec,           &
                                Node_type      Node)

!+
! This function compares the string described by Key_string with
! the string contained in the data node Node, and returns 0
! if the strings are equal, -1 if Key_string is < Node, and
! 1 if Key_string > Node.
!-

OPTION TYPE = EXPLICIT

RECORD Tree_element          ! Define the structure of our
  LONG   Seq_num             ! record. This record could
  LONG   Text_len           ! contain many useful data items
  STRING Text = 80
END RECORD Tree_element

RECORD Node_type
  BYTE   Header (9)         ! Header
  LONG   Seq_Num            ! Sequence number
  LONG   Length             ! Length
  STRING Text = 80         ! String
END RECORD Node_type

DECLARE STRING Node_string, Key_string

!+
! Return the result of the comparison.
!-

Node_string = SEG$ (Node::Text, 1, Node::Length)
Key_string = SEG$ (Rec::Text, 1, Rec::Text_len)

SELECT Key_string
CASE < Node_string
  Compare_node_2 = -1%
CASE > Node_string
  Compare_node_2 = 1%
CASE ELSE
  Compare_node_2 = 0%
END SELECT

END FUNCTION
```

# LIB\$INSERT\_TREE

```
%TITLE 'LIB$ Tree Example in BASIC V2'
%SBTTL 'Function to display node data'
400 SUB Display_node (                               &
      Node_type   Node )

!+
!   This routines prints the data into the node of the tree
!   once LIB$LOOKUP_TREE has been called to find the node
!-

      RECORD Node_type
          BYTE      Header (9)                ! Header
          LONG      Seq_Num                    ! Sequence number
          LONG      Length                     ! Length
          STRING    Text = 80                  ! String
      END RECORD Node_type

DECLARE STRING Node_string
Node_string = SEG$ (Node::Text, 1, Node::Length)
PRINT "The sequence number for ";Node_string;" is ";Node::Seq_num
END SUB
```

This BASIC example illustrates the use of LIB\$INSERT\_TREE, LIB\$LOOKUP\_TREE, and LIB\$TRAVERSE\_TREE.

The output generated by this program is as follows:

```
$ run tree
Enter one word per line, ^Z to begin searching the tree
> apple
> orange
> peach
> pear
> grapefruit
> lemon
> CTRL/Z
```

You will now be prompted for words to find. Enter one per line.

```
Word to find? lime
The word you entered does not appear in the tree
```

```
Word to find? orange
The sequence number for "orange" is 2
```

```
Word to find? CTRL/Z
```

The following is a dump of the tree. Notice that the words are in alphabetical order

```
1         apple
5         grapefruit
6         lemon
2         orange
3         peach
4         pear
$
```

# LIB\$INSQHI

---

## LIB\$INSQHI Insert Entry at Head of Queue

The Insert Entry at Head of Queue routine inserts a queue entry at the head of the specified self-relative interlocked queue. LIB\$INSQHI makes the VMS INSQHI instruction available as a callable routine.

---

**FORMAT**            **LIB\$INSQHI** *entry ,header [,retry-count]*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:           **longword (unsigned)**  
                          access:          **write only**  
                          mechanism:       **by value**

---

**ARGUMENTS**        ***entry***  
                          VMS usage: **unspecified**  
                          type:           **unspecified**  
                          access:          **modify**  
                          mechanism:       **by reference, array reference**

Entry to be inserted by LIB\$INSQHI. The **entry** argument contains the address of this signed quadword-aligned array that must be at least eight bytes long. Bytes following the first eight bytes can be used for any purpose by the calling program.

***header***  
VMS usage: **quadword\_signed**  
type:           **quadword integer (signed)**  
access:          **modify**  
mechanism:       **by reference**

Queue header specifying the queue into which **entry** is to be inserted. The **header** argument contains the address of this signed aligned quadword integer. **Header** must be initialized to zero before first use of the queue; zero means an empty queue.

***retry-count***  
VMS usage: **longword\_unsigned**  
type:           **longword (unsigned)**  
access:          **read only**  
mechanism:       **by reference**

The number of times the insertion is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The **retry-count** argument is the address of an unsigned longword that contains the retry count value. A value of 1 causes no retries. The default value is 10.

---

**DESCRIPTION**

The queue into which LIB\$INSQHI inserts an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address. Two longwords, a forward link and a backward link, define the location of the entry in relation to the preceding and succeeding entries.

A self-relative queue is a queue in which the links between entries are displacements; the two longwords represent the displacements of the current entry's predecessor and successor. The VMS instructions INSQHI, INSQTI, REMQHI, and REMQTI allow you to insert and remove an entry at the head or tail of a self-relative queue. Each queue instruction has a corresponding Run-Time Library routine.

The self-relative queue instructions are interlocked and cannot be interrupted, so that other processes cannot insert or remove queue entries while the current program is doing so. Since the operation requires changing two pointers at the same time, a high-level language cannot perform this operation without calling the Run-Time Library queue Access Routines.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue Access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an asynchronous system trap.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed. The entry was added to the front of the queue, and the resulting queue contains more than one entry.
LIB\$_ONEENTQUE	Routine successfully completed. The entry was added to the front of the queue, and the resulting queue contains one entry.
LIB\$_SECINTFAI	A secondary interlock failure occurred; the insertion was attempted the number of times specified by <b>retry-count</b> . This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.

---

**EXAMPLES**

```

1  INTEGER*4 FUNCTION INSERT_Q (QENTRY)
      COMMON/QUEUES/QHEADER
      INTEGER*4 QENTRY(10), QHEADER(2)
      INSERT_Q = LIB$INSQHI (QENTRY, QHEADER)
      RETURN
      END

```

This is a FORTRAN application using processor-shared memory.

# LIB\$INSQHI

```
2  COM (QUEUES) QENTRY%(9), QHEADER%(1)
   EXTERNAL INTEGER FUNCTION LIB$INSQHI
   IF LIB$INSQHI (QENTRY%() BY REF, QHEADER%() BY REF) AND 1%
       THEN GOTO 1000
       .
       .
       .
1000 REM  INSERTED OK
```

In BASIC and FORTRAN, queues can be quadword aligned in a named COMMON block by using a linker option file to specify PSECT alignment. The Run-Time Library routine LIB\$GET\_VM returns memory that is quadword aligned. Therefore, you should use LIB\$GET\_VM to allocate the virtual memory for a queue. For instance, to create a COMMON block called QUEUES, use the LINK command with the FILE/OPTIONS qualifier, where FILE.OPT is a linker option file containing the line:

```
PSECT = QUEUES, QUAD
```

---

## LIB\$INSQTI Insert Entry at Tail of Queue

The Insert Entry at Tail of Queue routine inserts a queue entry at the tail of the specified self-relative interlocked queue. LIB\$INSQTI makes the VAX INSQTI instruction available as a callable routine.

---

**FORMAT**            **LIB\$INSQTI** *entry ,header [,retry-count]*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        ***entry***  
                           VMS usage: **unspecified**  
                           type:        **unspecified**  
                           access:      **modify**  
                           mechanism: **by reference, array reference**

Entry to be inserted at the tail of the queue by LIB\$INSQTI. The **entry** argument contains the address of this signed quadword-aligned array that must be at least eight bytes long. Bytes following the first eight bytes can be used for any purpose by the calling program.

### ***header***

VMS usage: **quadword\_signed**  
 type:        **quadword integer (signed)**  
 access:     **modify**  
 mechanism: **by reference**

Queue header specifying the queue into which the queue entry is to be inserted. The **header** argument contains the address of this signed aligned quadword integer. **Header** must be initialized to zero before first use of the queue; zero means an empty queue.

### ***retry-count***

VMS usage: **longword\_unsigned**  
 type:        **longword (unsigned)**  
 access:     **read only**  
 mechanism: **by reference**

The number of times the insertion is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The **retry-count** argument is the address of a longword which contains the retry count value. The default value is 10.

# LIB\$INSQTI

---

## DESCRIPTION

The queue into which LIB\$INSQTI inserts an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address. Two longwords, a forward link and a backward link, define the location of the entry in relation to the preceding and succeeding entries.

A self-relative queue is a queue in which the links between entries are displacements; the two longwords represent the displacements of the current entry's predecessor and successor. The VAX instructions INSQHI, INSQTI, REMQHI, and REMQTI allow you to insert and remove an entry at the head or tail of a self-relative queue. Each queue instruction has a corresponding Run-Time Library routine.

The self-relative queue instructions are interlocked and cannot be interrupted, so that other processes cannot insert or remove queue entries while the current program is doing so. Since the operation requires changing two pointers at the same time, a high-level language cannot perform this operation without calling the Run-Time Library queue access routines.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue Access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an asynchronous system trap.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. The entry was added to the tail of the queue: the resulting queue contains more than one entry.
LIB\$_ONEENTQUE	Routine successfully completed. The entry was added to the tail of the queue: the resulting queue contains one entry.
LIB\$_SECINTFAI	A secondary interlock failure occurred; the insertion was attempted the number of times specified by <b>retry-count</b> . This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.





# LIB\$INSV

## ***base-address***

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by value**

Field into which LIB\$INSV writes the source field. The **base-address** argument is an unsigned longword containing the base address of this aligned bit string.

---

## **CONDITION VALUE SIGNALLED**

SS\$\_ROPRAND

A reserved operand fault is signaled if a size greater than 32 is specified.

---

## **EXAMPLES**

**1** INTEGER\*4 COND\_VALUE  
CALL LIB\$INSV (4, 0, 3, COND\_VALUE)

This example shows how to set bits 0 through 2 of longword COND\_VALUE to the value 4 in FORTRAN.

**2** DECLARE INTEGER COND\_VALUE  
CALL LIB\$INSV (4%, 0%, 3%, COND\_VALUE)

This example uses BASIC to set bits 0 through 2 of longword COND\_VALUE to the value 4.

---

## LIB\$INT\_OVER Integer Overflow Detection

The Integer Overflow Detection routine enables or disables integer overflow detection for the calling routine activation. The previous integer overflow enable setting is returned.

---

**FORMAT**            **LIB\$INT\_OVER** *new-setting*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

Old integer overflow enable setting (the previous contents of SF\$W\_PSW[PSW\$V\_IV] in the caller's frame).

---

**ARGUMENT**            *new-setting*  
                           VMS usage: **longword\_unsigned**  
                           type:            **longword (unsigned)**  
                           access:        **read only**  
                           mechanism:    **by reference**

New integer overflow enable setting. The **new-setting** argument is the address of an unsigned longword which contains the new integer overflow enable setting. Bit 0 set to 1 means enable, bit 0 set to 0 means disable.

---

**DESCRIPTION**        The caller's stack frame will be modified by this routine.

LIB\$INT\_OVER affects only the current routine activation and does not affect any of its callers or any routines that it may call. However, the setting remains in effect for any routines which are subsequently entered through a JSB entry point.

---

**CONDITION  
VALUES  
RETURNED**            *None.*

# LIB\$INT\_OVER

---

## EXAMPLE

```
INTOVF: ROUTINE OPTIONS (MAIN);
DECLARE LIB$INT_OVER ENTRY (FIXED BINARY (7)) /* Address of byte for
/* enable/disable
/* setting */
        RETURNS (FIXED BINARY (31)); /* Old setting */
DECLARE DISABLE FIXED BINARY (7) INITIAL (0) STATIC READONLY;
DECLARE (A,B) FIXED BINARY (7);

ON FIXEDOVERFLOW PUT SKIP LIST ('Overflow');

A = 127;
B = A + 2;
PUT LIST ('In MAIN');

        BEGIN;

        DECLARE RESULT FIXED BINARY (31);
/* Disable recognition of integer overflow in this block */
        RESULT = LIB$INT_OVER (DISABLE);

        B = A + 2;
        PUT SKIP LIST ('In BEGIN block');

        CALL Q;

                Q: routine;
                B = A + 2;
                PUT LIST ('In Q');
                END Q;

        END /* Begin */;

END INTOVF;
```

This PL/I routine shows how to use LIB\$INT\_OVER to enable or disable the detection of integer overflow. Note that in PL/I integer overflow is always enabled unless explicitly overridden by a call to this routine. However, disabling integer overflow is only effective for the block which calls this routine; descendent blocks are unaffected. The output generated by this PL/I program is as follows:

```
        In MAIN
        In BEGIN block
        Overflow          In Q
```

---

## LIB\$LEN Length of String Returned as Longword Value

The Length of String Returned as Longword Value routine returns the length of a string.

---

**FORMAT**            **LIB\$LEN** *source-string*

---

**RETURNS**            VMS usage: **word\_unsigned**  
                         type:            **word (unsigned)**  
                         access:        **write only**  
                         mechanism: **by value**

Length of the source string, extracted and zero-extended to 32 bits.

---

**ARGUMENT**            ***source-string***  
                         VMS usage: **char\_string**  
                         type:            **character string**  
                         access:        **read only**  
                         mechanism: **by descriptor**

Source string whose length is returned by LIB\$LEN. The **source-string** argument contains the address of a descriptor pointing to this source string.

---

**DESCRIPTION**        The maximum length of a VMS string is 65,535 characters.

The BASIC and FORTRAN intrinsic function LEN generates equivalent in-line code at run time. Therefore, it is more efficient for BASIC and FORTRAN users to use the intrinsic function LEN than to call LIB\$LEN.

If you need both the length of the string and the address of its first byte, you should use LIB\$ANALYZE\_SDESC instead.

---

**CONDITION  
VALUES  
RETURNED**            *None.*

# LIB\$LOCC

---

## LIB\$LOCC Locate a Character

The Locate a Character routine locates a character in a string by comparing successive bytes in the string with the character specified. The search continues until the character is found or the string has no more characters. LIB\$LOCC makes the VAX LOCC instruction available as a callable routine.

---

**FORMAT**            **LIB\$LOCC** *character-string ,source-string*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

The relative position from the start of **source-string** to the first equal character or zero if no match is found.

---

**ARGUMENTS**        ***character-string***  
                          VMS usage: **char\_string**  
                          type:        **character string**  
                          access:     **read only**  
                          mechanism: **by descriptor**

String whose initial character is used by LIB\$LOCC in the search. The **character-string** argument contains the address of a descriptor pointing to this string. Only the first character of **character-string** is used, and its length is not checked.

***source-string***  
VMS usage: **char\_string**  
type:        **character string**  
access:     **read only**  
mechanism: **by descriptor**

String to be searched by LIB\$LOCC. The **source-string** argument is the address of a descriptor pointing to this character string.

---

**DESCRIPTION**        LIB\$LOCC returns the position of the first equal character relative to the start of the source string as an index. An index is the relative position of the first occurrence of a substring in the source string. If no character matches, or if the string has a length of zero, then a zero is returned, indicating that the character was not found.

---

**CONDITION**        *None.*  
**VALUES**  
**RETURNED**

---

**EXAMPLES**
**1**

```

IDENTIFICATION DIVISION.
PROGRAM-ID.          LIBLOC.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01  SEARCH-STRING  PIC X(26)
                               VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
01  SEARCH-CHAR    PIC X.
01  IND-POS        PIC 9(9) USAGE IS COMP.
01  DISP-IND       PIC 9(9).

ROUTINE DIVISION.

001-MAIN.
    MOVE SPACE TO SEARCH-CHAR.
    DISPLAY " ".
    DISPLAY "ENTER SEARCH CHARACTER: " WITH NO ADVANCING.
    ACCEPT SEARCH-CHAR.
    CALL "LIB$LOCC"
        USING BY DESCRIPTOR SEARCH-CHAR, SEARCH-STRING
        GIVING IND-POS.
    IF IND-POS = ZERO
        DISPLAY
            "CHAR ENTERED (" SEARCH-CHAR ") NOT A VALID SEARCH CHAR"
        STOP RUN.
    MOVE IND-POS TO DISP-IND.
    DISPLAY
        "SEARCH CHAR (" SEARCH-CHAR ") WAS FOUND IN POSITION "
        DISP-IND.
    GO TO 001-MAIN.

```

This COBOL program accepts a character as input and returns as output the character's position in a search string. The output generated by this COBOL program is as follows:

```

$ RUN LIBLOC
ENTER SEARCH CHARACTER: X
SEARCH CHAR (X) WAS FOUND IN POSITION 00000024

ENTER SEARCH CHARACTER: Y
SEARCH CHAR (Y) WAS FOUND IN POSITION 00000025

ENTER SEARCH CHARACTER: B
SEARCH CHAR (B) WAS FOUND IN POSITION 00000002

ENTER SEARCH CHARACTER: b
CHAR ENTERED (b) NOT A VALID SEARCH CHAR
$

```

Notice that uppercase and lowercase letters are not considered equal.

# LIB\$LOCC

```
2 10  !+
      ! This is an BASIC program demonstrating the
      ! use of LIB$LOCC.
      !-

      EXTERNAL INTEGER FUNCTION LIB$LOCC
      I% = 0
      CHARSTR$ = 'DAY'
      SRCSTR$ = 'ONE DAY AT A TIME'
      I% = LIB$LOCC(CHARSTR$, SRCSTR$)
      PRINT I%
90  END
```

This BASIC example also illustrates the use of LIB\$LOCC. The output generated by this BASIC program is "5".





# LIB\$LOOKUP\_KEY

## ***keyword-string***

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

Full keyword match. The **keyword-string** argument contains the address of a descriptor pointing to the keyword string. LIB\$LOOKUP\_KEY writes the address of this descriptor into **keyword-string** if the full keyword is matched.

## ***resultant-length***

VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **write only**  
mechanism: **by reference**

Number of characters in the keyword, independent of padding. The **resultant-length** argument is the address of an unsigned word integer that contains the number of characters in the keyword. LIB\$LOOKUP\_KEY writes the address of this signed word integer into **resultant-length**.

---

## **DESCRIPTION**

LIB\$LOOKUP\_KEY is intended to help programmers to write utilities that have command qualifiers with values.

LIB\$LOOKUP\_KEY locates a matching keyword or keyword abbreviation by comparing the first *n* characters of each keyword in the keyword table with the supplied string, where *n* is the length of the supplied string.

When a keyword match is found, the following information is optionally returned to the caller.

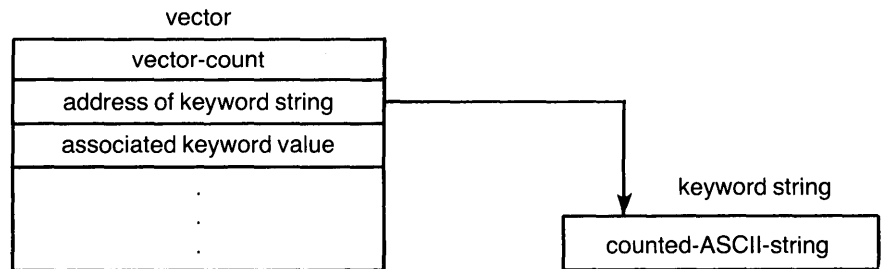
- The longword value associated with the matched keyword
- The full keyword string (any descriptor type)

An exact match is found if the length of the keyword found is equal to the length of the supplied string.

If an exact keyword match is found, no further processing is performed, and a normal return status is returned to the caller. Otherwise, after a match has been found, the rest of the keyword table is scanned. If an additional match is found, a "not enough characters" return status is returned to the caller. If the keyword table contains a keyword that is an abbreviation of another keyword in the table, an exact match can occur for short abbreviations.

See Figure LIB-5 for the structure of the keyword table, which the calling program creates for this routine.

Figure LIB-5 Keyword Table



ZK-1976-84

**Vector-count** is the number of longwords that follow, and **counted-ASCII-string** starts with a byte that is the unsigned count of the number of ASCII characters that follow.

Because of the format of the keyword table, this routine cannot be called easily from high-level languages. The examples show how to use a macro, `$LIB_KEY_TABLE`, to construct a keyword table from MACRO or BLISS. A separate example shows how a table could be constructed in FORTRAN.

Use of the `$LIB_KEY_TABLE` macro results in data that is not position-independent code (PIC). If your application requires PIC data, you must fill in the address of the keyword strings at execution time. See the FORTRAN example for a demonstration of this technique.

### CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. A unique keyword match was found.
LIB\$_AMBKEY	Multiple keyword match found. Not enough characters were specified to allow a unique match.
LIB\$_INVARG	Invalid arguments, not enough arguments, and/or bad keyword table.
LIB\$_INSVIRMEM	Insufficient virtual memory to return keyword string. This is only possible if <b>keyword-string</b> is a dynamic string.
LIB\$_STRTRU	String truncated.
LIB\$_UNRKEY	The keyword you specified does not appear in the keyword table you specified.

# LIB\$LOOKUP\_KEY

---

## EXAMPLES

**1** KEYTABLE:  
    \$LIB\_KEY\_TABLE < -  
        <ADD, 1>, -  
        <DELETE, 2>, -  
        <EXIT, 3>>

This MACRO fragment defines a keyword table named KEYTABLE containing the three keywords ADD, DELETE, and EXIT with associated keyword values of 1, 2, and 3, respectively.

The \$LIB\_KEY\_TABLE macro is supplied in the default macro library SYS\$LIBRARY:STARLET.MLB. Because this library is automatically searched by the assembler, you do not have to specify it in the DCL command MACRO.

**2** LIBRARY 'SYS\$LIBRARY:STARLET.L32';  
OWN  
    KEYTABLE: \$LIB\_KEY\_TABLE (  
        (ADD, 1),  
        (DELETE, 2),  
        (EXIT, 3));

This BLISS code fragment specifies that SYS\$LIBRARY:STARLET.L32 is to be searched to resolve references. It defines a keyword table named KEYTABLE containing the three keywords ADD, DELETE, and EXIT with associated keyword values of 1, 2, and 3, respectively.

The \$LIB\_KEY\_TABLE macro is supplied in the BLISS library SYS\$LIBRARY:STARLET.L32 and in the BLISS require file SYS\$LIBRARY:STARLET.REQ. BLISS does not automatically search either of these files so you must explicitly cause them to be searched by including the appropriate LIBRARY or REQUIRE statement in your module. You should use the precompiled library because it is more efficient for the compiler.

**3** PARAMETER (  
1    MAXKEYSIZE = 6,                    ! Maximum keyword size  
2    NKEYS = 3)                        ! Number of keywords  
BYTE KEYWORDS (MAXKEYSIZE+1, NKEYS)  
INTEGER\*4 KEYTABLE (0:NKEYS\*2)  
DATA KEYWORDS /  
1    3, 'A', 'D', 'D', ' ', ' ', ' ', ' ',                    ! Counted ASCII 'ADD'  
2    6, 'D', 'E', 'L', 'E', 'T', 'E', ' ',                    ! Counted ASCII 'DELETE'  
3    4, 'E', 'X', 'I', 'T', ' ', ' ', ' ',                    ! Counted ASCII 'EXIT'  
  
KEYTABLE(0) = NKEYS\*2                    ! Number of longwords to follow  
KEYTABLE(1) = %LOC(KEYWORDS(1,1))        ! Address of keyword string  
KEYTABLE(2) = 1                         ! Keyword value for 'ADD'  
KEYTABLE(3) = %LOC(KEYWORDS(1,2))        ! Address of keyword string  
KEYTABLE(4) = 2                         ! Keyword value for 'DELETE'  
KEYTABLE(5) = %LOC(KEYWORDS(1,3))        ! Address of keyword string  
KEYTABLE(6) = 3                         ! Keyword value for 'EXIT'

This FORTRAN code fragment constructs a keyword table named KEYTABLE containing the three keywords ADD, DELETE, and EXIT with associated keyword values of 1, 2, and 3, respectively. This construction method results in position-independent coded data, although the generated code for the typical FORTRAN module contains other non-PIC values.



# LIB\$LOOKUP\_TREE

---

Value	Description
Negative	<b>symbol</b> is less than the current node
Zero	<b>symbol</b> is equal to the current node
Positive	<b>symbol</b> is greater than the current node

---

For more information on the compare routine, see "Call Format for a Compare Routine" in the Description section.

## ***new-node***

VMS usage: **address**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by reference**

Location where the new symbol was found. The **new-node** argument is the address of an unsigned longword that is the new node location.

---

## DESCRIPTION

### Call Format for a Compare Routine

The call format for a compare routine is as follows:

```
user-compare-routine  symbol ,treehead
```

LIB\$LOOKUP\_TREE passes the **symbol** and **treehead** arguments to the compare routine using the same passing mechanism that was used to pass them to LIB\$LOOKUP\_TREE.

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Success. The key was found.
LIB\$_KEYNOTFOU	Error. The key was not found.

---

## EXAMPLE

The BASIC example provided in the description of LIB\$INSERT\_TREE also demonstrates how to use LIB\$LOOKUP\_TREE. Please refer to that example for assistance in using this routine.

---

## LIB\$LP\_LINES Lines on Each Printer Page

The Lines on Each Printer Page routine computes the default number of lines on a printer page. This routine can be used by native-mode VMS utilities that produce listing files and paginate files.

---

### FORMAT LIB\$LP\_LINES

---

#### RETURNS

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **write only**  
mechanism: **by value**

The default number of lines on a physical printer page. If the logical name translation or conversion to binary fails, a default value of 66 is returned.

---

#### ARGUMENTS *None.*

---

#### DESCRIPTION

LIB\$LP\_LINES computes the default number of lines on a printer page. This routine can be used by native-mode VMS utilities that produce listing files and paginate files. The algorithm used by LIB\$LP\_LINES is:

- 1 Translate the logical name SYS\$LP\_LINES.
- 2 Convert the ASCII value obtained to a binary integer.
- 3 Verify that the resulting value is in the range [30:255].
- 4 If any of the prior steps fail, return the default paper size of 66 lines.

You can use LIB\$LP\_LINES to monitor the current default length of the line printer page. You can also supply your own default length for the current process. United States standard paper stock permits 66 lines on each physical page.

If you are writing programs for a utility that formats a listing file to be printed on a line printer, you can use LIB\$LP\_LINES to make your utility independent of the default page length. Your program can use LIB\$LP\_LINES to obtain the current length of the page. It can then calculate the number of lines of text on each page by subtracting the lines used for margins and headings.

The following is one suggested format.

- 1 Three lines for the top margin
- 2 Three lines for the bottom margin

# LIB\$LP\_LINES

- 3 Three lines for listing heading information, consisting of:
  - a. A language-processor identification line
  - b. A source-program identification line
  - c. One blank line

---

**CONDITION VALUES RETURNED**      *None.*

---

## EXAMPLES

**1**

```
lplines = LIB$LP_LINES()
PRINT 10, lplines
10  Format (' Line printer page = ',I5,' lines.')
```

end

This FORTRAN program displays the current default length of the line printer page.

**2**

```
LINES: ROUTINE OPTIONS (MAIN);
  DECLARE LIB$LP_LINES EXTERNAL ENTRY
  RETURNS (FIXED BINARY (31));
  PUT SKIP LIST ('Line printer page = ',LIB$LP_LINES(),' lines.');
```

END;

This PL/I program displays the current default length of the line printer page.

**3**

```
100 EXTERNAL INTEGER FUNCTION LIB$LP_LINES
200 DECLARE INTEGER LPLINES
300 LPLINES = LIB$LP_LINES
400 PRINT "Line printer page = "; LPLINES
32767 END
```

This BASIC program displays the current default length of the line printer page.

**4**

```
PROGRAM LINES(OUTPUT);
FUNCTION LIB$LP_LINES : INTEGER;
  EXTERN;

BEGIN
  WRITELN('Line printer page = ',LIB$LP_LINES,' lines.');
```

END.

This Pascal program displays the current default length of the line printer page.

# LIB\$LP\_LINES

5

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. PAGELINES.  
  
DATA DIVISION.  
  
WORKING-STORAGE SECTION.  
01 LPLINES PIC 9(9) USAGE IS COMP  
                                VALUE IS 999999999.  
01 SHOWPLINES PIC 9(9).  
  
ROUTINE DIVISION.  
PO.  
    CALL "LIB$LP_LINES"  
    GIVING LPLINES.  
MOVE LPLINES TO SHOWPLINES.  
DISPLAY "Line printer page = ", SHOWPLINES, " lines."  
STOP RUN.
```

This COBOL program displays the current default length of the line printer page.



# LIB\$MATCHC

---

## LIB\$MATCHC Match Characters, Return Relative Position

The Match Characters and Return Relative Position routine searches a source string for a specified substring and returns an index, which is the relative position of the first occurrence of a substring in the source string. The relative character positions returned by LIB\$MATCHC are numbered 1, 2, . . . , n. Thus, zero means that the substring was not found. LIB\$MATCHC makes the VAX MATCHC instruction available as a callable routine.

---

**FORMAT**            **LIB\$MATCHC** *sub-string,source-string*

---

**RETURNS**            VMS usage: **longword\_unsigned**  
                          type:            **longword (unsigned)**  
                          access:        **write only**  
                          mechanism:    **by value**

The relative position of the first character of the substring if found, or zero if not found.

---

**ARGUMENTS**        ***sub-string***  
                          VMS usage: **char\_string**  
                          type:         **character string**  
                          access:       **read only**  
                          mechanism:   **by descriptor**

Substring to be found. The **sub-string** argument is the address of a descriptor pointing to this substring.

***source-string***  
VMS usage: **char\_string**  
type:        **character string**  
access:      **read only**  
mechanism:  **by descriptor**

Source string to be searched by LIB\$MATCHC. The **source-string** argument is the address of a descriptor pointing to this source string.

---

**DESCRIPTION**      LIB\$MATCHC searches a source string for a specified substring and returns an index, which is the relative position of the first occurrence of a substring in the source string.

The relative character positions returned by LIB\$MATCHC are numbered 1, 2, . . . , n. Thus, zero means that the substring was not found.

If the substring has a zero length, LIB\$MATCHC returns the value 1, indicating success, no matter how long the source string is. If the source string has a zero length and the substring has a nonzero length, zero is returned, indicating that the substring was not found.

# LIB\$MATCHC

The order of arguments for LIB\$MATCHC parallels the VAX MATCHC instruction.

---

**CONDITION  
VALUES  
RETURNED**

*None.*



# LIB\$MATCH\_COND

specific), the facility code field (STS\$V\_FAC\_NO = bits 27:17) is ignored and only the STS\$V\_MSG\_ID fields (bits 15:3) are compared.

The routine returns a zero if a match is not found, a 1 if the condition value matches the first condition value in the list (the second argument), a 2 if it matches the second condition value (the third argument), and so on. LIB\$MATCH\_COND checks for null argument entries in the argument list.

When LIB\$MATCH\_COND is called with only two arguments, the possible values for the value returned are true (1) or false (zero).

Each condition handler must examine the signal argument vector to determine which condition is being signaled. If the condition is not one that the handler knows about, the handler should resignal. A handler should not assume that only one kind of condition can occur in the routine which established it or in any routines it calls. However, because a condition value may be modified by an intervening handler, each handler should only compare that part of the condition value that distinguishes it from another.

---

<b>CONDITION VALUES RETURNED</b>	<i>None.</i>
----------------------------------	--------------

---

## EXAMPLE

```
C+
C   This FORTRAN program demonstrates the use of
C   LIB$MATCH_COND.
C
C   Declare handler routine as external.
C-
      EXTERNAL      HANDLER
C+
C   Declare the handler that will be used.
C-
      TYPE * , 'Establishing handler...'
      CALL LIB$ESTABLISH( HANDLER )
      OPEN ( UNIT = 1 , NAME = 'MATCH.DAT' , STATUS = 'OLD' )
C+
C   Revert to normal error processing.
C-
      CALL LIB$REVERT
      CLOSE ( UNIT = 1 )
      CALL EXIT
      END
C+
C   This is the handler routine.
C-
      INTEGER*4 FUNCTION HANDLER( SIGARGS , MECHARGS )
      INTEGER*4 MECHARGS(*) , SIGARGS(*) , STATUS
      INCLUDE '$SSDEF'
      INCLUDE '$FORDEF'
      HANDLER = SS$_CONTINUE
```

# LIB\$MATCH\_COND

```
C+
C   This handler will type out an error message.  In this case the
C   message is regarding a file open status.
C-

      TYPE * , 'Entering handler...'
      STATUS = LIB$MATCH_COND( SIGARGS( 2 ) , FOR$_FILNOTFOU ,
1      FOR$_NO_SUCDEV , FOR$_FILNAMSPE , FOR$_OPEFAI )
      GOTO ( 100 , 200 , 300 , 400 ) STATUS
      HANDLER = SS$_RESIGNAL
      GOTO 1000
100   TYPE * , 'ERROR -- File not found'
      GOTO 1000
200   TYPE * , 'ERROR -- No such Device'
      GOTO 1000
300   TYPE * , 'ERROR -- File name specification'
      GOTO 1000
400   TYPE * , 'ERROR -- Open Failure'
1000  CALL SYS$UNWIND( MECHARGS( 3 ) , )
      TYPE * , 'Returning from handler...'
      RETURN
      END
```

This FORTRAN program uses a computed GOTO to alter the program execution sequence on a condition value.

If the file called MATCH.DAT does not exist, the following output is returned:

```
Establishing handler...
Entering handler...
ERROR -- File not found
Returning from handler...
```

If the file MATCH.DAT does exist, the output returned is as follows:

```
Establishing handler...
```

---

## LIB\$MOV3 Move Characters

The Move Characters routine makes the VAX MOV3 instruction available as a callable routine. The source item is moved to the destination item. Overlap of the source and destination items does not affect the result.

---

**FORMAT**            **LIB\$MOV3** *word-integer-length ,source ,destination*

---

**RETURNS**            None.

---

**ARGUMENTS**        ***word-integer-length***  
 VMS usage: **word\_unsigned**  
 type:            **word (unsigned)**  
 access:         **read only**  
 mechanism:     **by reference**

Number of bytes to be moved from **source** to **destination** by LIB\$MOV3. The **word-integer-length** argument is the address of an unsigned word which contains this number of bytes. The maximum transfer is 65,535 bytes.

***source***  
 VMS usage: **unspecified**  
 type:            **unspecified**  
 access:         **read only**  
 mechanism:     **by reference**

Item to be moved. The **source** argument is the address of this item.

***destination***  
 VMS usage: **unspecified**  
 type:            **unspecified**  
 access:         **write only**  
 mechanism:     **by reference**

Item into which **source** will be moved. The **destination** argument is the address of this item.

---

**DESCRIPTION**        LIB\$MOV3 is useful for moving large blocks of data, such as arrays, when such an operation would otherwise have to be performed by a programmed loop.

For more information, see the *VAX Architecture Reference Manual*. See also OTS\$MOVE3.

---

**CONDITION  
VALUES  
RETURNED**            None.



of bytes. The maximum value of **word-integer-destination-length** is 65,535 bytes.

## ***destination***

VMS usage: **unspecified**  
type: **unspecified**  
access: **write only**  
mechanism: **by reference**

Item into which **source** will be moved. The **destination** argument is the address of this item.

---

## **DESCRIPTION**

If the destination item is shorter than the source item, the highest-addressed bytes of the source are not moved.

For more information, see the *VAX Architecture Reference Manual*. See also OTS\$MOVE5.

---

## **CONDITION VALUES RETURNED**

*None.*





the Description section illustrates the creation of a string descriptor for a translation table using VAX BASIC.

## ***destination-string***

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

Destination string into which LIB\$MOVTC writes the translated **source-string**. The **destination-string** argument is the address of a descriptor pointing to this destination string.

---

## **DESCRIPTION**

Each character in the source string is used as an index into the translation table. The byte found is then placed into the destination string. The fill character is used if the destination string is longer than the source string. If the source string is longer than the destination string, the source string is truncated. Overlap of the source and destination strings does not affect execution.

The translation tables used by LIB\$MOVTC and LIB\$MOVTUC are described below. Each table is preceded by explanatory text.

### **ASCII to EBCDIC Translation Table**

- The number on the left represents the low-order bits of the ASCII character in hexadecimal notation.
- The number across the top represents the high-order bits of the ASCII character in hexadecimal notation.
- The number in the body of the table represents the equivalent EBCDIC character in hexadecimal notation.

# LIB\$MOVTC

**Table LIB-7 LIB\$AB\_ASC\_EBC**

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	40	F0	7C	D7	79	97	3F	3F	3F	3F	3F	3F	3F	3F
1	01	11	4F	F1	C1	D8	81	98	3F	3F	3F	3F	3F	3F	3F	3F
2	02	12	7F	F2	C2	D9	82	99	3F	3F	3F	3F	3F	3F	3F	3F
3	03	13	7B	F3	C3	E2	83	A2	3F	3F	3F	3F	3F	3F	3F	3F
4	37	3C	5B	F4	C4	E3	84	A3	3F	3F	3F	3F	3F	3F	3F	3F
5	2D	3D	6C	F5	C5	E4	85	A4	3F	3F	3F	3F	3F	3F	3F	3F
6	2E	32	50	F6	C6	E5	86	A5	3F	3F	3F	3F	3F	3F	3F	3F
7	2F	26	7D	F7	C7	E6	87	A6	3F	3F	3F	3F	3F	3F	3F	3F
8	16	18	4D	F8	C8	E7	88	A7	3F	3F	3F	3F	3F	3F	3F	3F
9	05	19	5D	F9	C9	E8	89	A8	3F	3F	3F	3F	3F	3F	3F	3F
A	25	3F	5C	7A	D1	E9	91	A9	3F	3F	3F	3F	3F	3F	3F	3F
B	0B	27	4E	5E	D2	4A	92	C0	3F	3F	3F	3F	3F	3F	3F	3F
C	0C	1C	6B	4C	D3	E0	93	6A	3F	3F	3F	3F	3F	3F	3F	3F
D	0D	1D	60	7E	D4	5A	94	D0	3F	3F	3F	3F	3F	3F	3F	3F
E	0E	1E	4B	6E	D5	5F	95	A1	3F	3F	3F	3F	3F	3F	3F	3F
F	0F	1F	61	6F	D6	6D	96	07	3F	3F	3F	3F	3F	3F	3F	FF

ZK-4246-85

## ASCII to EBCDIC Reversible Translation Table

- The number on the left represents the low-order bits of the ASCII character in hexadecimal notation.
- The number across the top represents the high-order bits of the ASCII character in hexadecimal notation.
- The number in the body of the table represents the equivalent EBCDIC character in hexadecimal notation.

**Table LIB-8 LIB\$AB\_ASC\_EBC\_REV**

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	40	F0	7C	D7	79	97	20	30	41	58	76	9F	B8	DC
1	01	11	4F	F1	C1	D8	81	98	21	31	42	59	77	A0	B9	DD
2	02	12	7F	F2	C2	D9	82	99	22	1A	43	62	78	AA	BA	DE
3	03	13	7B	F3	C3	E2	83	A2	23	33	44	63	80	AB	BB	DF
4	37	3C	5B	F4	C4	E3	84	A3	24	34	45	64	8A	AC	BC	EA
5	2D	3D	6C	F5	C5	E4	85	A4	15	35	46	65	8B	AD	BD	EB
6	2E	32	50	F6	C6	E5	86	A5	06	36	47	66	8C	AE	BE	EC
7	2F	26	7D	F7	C7	E6	87	A6	17	08	48	67	8D	AF	BF	ED
8	16	18	4D	F8	C8	E7	88	A7	28	38	49	68	8E	B0	CA	EE
9	05	19	5D	F9	C9	E8	89	A8	29	39	51	69	8F	B1	CB	EF
A	25	3F	5C	7A	D1	E9	91	A9	2A	3A	52	70	90	B2	CC	FA
B	0B	27	4E	5E	D2	4A	92	C0	2B	3B	53	71	9A	B3	CD	FB
C	0C	1C	6B	4C	D3	E0	93	6A	2C	04	54	72	9B	B4	CE	FC
D	0D	1D	60	7E	D4	5A	94	D0	09	14	55	73	9C	B5	CF	FD
E	0E	1E	4B	6E	D5	5F	95	A1	0A	3E	56	74	9D	B6	DA	FE
F	0F	1F	61	6F	D6	6D	96	07	1B	E1	57	75	9E	B7	DB	FF

ZK-4248-85

### EBCDIC to ASCII Translation Table

- The number on the left represents the low-order bits of the EBCDIC character in hexadecimal notation.
- The number across the top represents the high-order bits of the EBCDIC character in hexadecimal notation.
- The number in the body of the table represents the equivalent ASCII character in hexadecimal notation.

# LIB\$MOVTC

**Table LIB-9 LIB\$AB\_EBC\_ASC**

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	5C	5C	20	26	2D	5C	5C	5C	5C	7B	7D	5C	30	
1	01	11	5C	5C	5C	5C	2F	5C	61	6A	7E	5C	41	4A	5C	31
2	02	12	5C	16	5C	5C	5C	5C	62	6B	73	5C	42	4B	53	32
3	03	13	5C	5C	5C	5C	5C	5C	63	6C	74	5C	43	4C	54	33
4	5C	5C	5C	5C	5C	5C	5C	5C	64	6D	75	5C	44	4D	55	34
5	09	5C	0A	5C	5C	5C	5C	5C	65	6E	76	5C	45	4E	56	35
6	5C	08	17	5C	5C	5C	5C	5C	66	6F	77	5C	46	4F	57	36
7	7F	5C	1B	04	5C	5C	5C	5C	67	70	78	5C	47	50	58	37
8	5C	18	5C	5C	5C	5C	5C	5C	68	71	79	5C	48	51	59	38
9	5C	19	5C	5C	5C	5C	5C	60	69	72	7A	5C	49	52	5A	39
A	5C	5C	5C	5C	5B	5D	7C	3A	5C	5C	5C	5C	5C	5C	5C	5C
B	0B	5C	5C	5C	2E	24	2C	23	5C	5C	5C	5C	5C	5C	5C	5C
C	0C	1C	5C	14	3C	2A	25	40	5C	5C	5C	5C	5C	5C	5C	5C
D	0D	1D	05	15	28	29	5F	27	5C	5C	5C	5C	5C	5C	5C	5C
E	0E	1E	06	5C	2B	3B	3E	3D	5C	5C	5C	5C	5C	5C	5C	5C
F	0F	1F	07	1A	21	5E	3F	22	5C	5C	5C	5C	5C	5C	5C	FF

ZK-4249-85

## EBCDIC to ASCII Reversible Translation Table

- The number on the left represents the low-order bits of the EBCDIC character in hexadecimal notation.
- The number across the top represents the high-order bits of the EBCDIC character in hexadecimal notation.
- The number in the body of the table represents the equivalent ASCII character in hexadecimal notation.

Table LIB-10 LIB\$AB\_EBC\_ASC\_REV

Row bits 0 - 3	Column bits 4 - 7																	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		
0	00	10	80	90	20	26	2D	BA	C3	CA	D1	D8	7B	7D	5C	30		
1	01	11	81	91	A0	A9	2F	BB	61	6A	7E	D9	41	4A	9F	31		
2	02	12	82	16	A1	AA	B2	BC	62	6B	73	DA	42	4B	53	32		
3	03	13	83	93	A2	AB	B3	BD	63	6C	74	DB	43	4C	54	33		
4	04	14	84	94	A3	AC	B4	BE	64	6D	75	DC	44	4D	55	34		
5	05	15	85	0A	95	A4	AD	B5	BF	65	6E	76	DD	45	4E	56	35	
6	06	16	86	08	17	96	A5	AE	B6	C0	66	6F	77	DE	46	4F	57	36
7	07	17	87	1B	04	A6	AF	B7	C1	67	70	78	DF	47	50	58	37	
8	08	18	88	98	A7	B0	B8	C2	68	71	79	E0	48	51	59	38		
9	09	19	89	99	A8	B1	B9	60	69	72	7A	E1	49	52	5A	39		
A	0A	1A	8A	9A	5B	5D	7C	3A	C4	CB	D2	E2	E8	EE	F4	FA		
B	0B	1B	8B	9B	2E	24	2C	23	C5	CC	D3	E3	E9	EF	F5	FB		
C	0C	1C	8C	14	3C	2A	25	40	C6	CD	D4	E4	EA	F0	F6	FC		
D	0D	1D	05	15	28	29	5F	27	C7	CE	D5	E5	EB	F1	F7	FD		
E	0E	1E	06	9E	2B	3B	3E	3D	C8	CF	D6	E6	EC	F2	F8	FE		
F	0F	1F	07	1A	21	5E	3F	22	C9	D0	D7	E7	ED	F3	F9	FF		

ZK-4250-85

### Packed Decimal to Trailing Overpunch Numeric Value Translation Table

- The number on the left represents the low-order bits of the packed decimal value in hexadecimal notation.
- The number across the top represents the high-order bits of the packed decimal value in hexadecimal notation.
- The number in the body of the table represents the equivalent trailing overpunch numeric value in hexadecimal notation.

# LIB\$MOVTC

**Table LIB-11 LIB\$AB\_CVTPT\_0**

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
1	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
2	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
3	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
4	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
5	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
6	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
7	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
8	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
9	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B	7B
A	7B	41	42	43	44	45	46	47	48	49	7B	7B	7B	7B	7B	7B
B	7D	4A	4B	4C	4D	4E	4F	50	51	52	7B	7B	7B	7B	7B	7B
C	7B	41	42	43	44	45	46	47	48	49	7B	7B	7B	7B	7B	7B
D	7D	4A	4B	4C	4D	4E	4F	50	51	52	7B	7B	7B	7B	7B	7B
E	7B	41	42	43	44	45	46	47	48	49	7B	7B	7B	7B	7B	7B
F	7B	41	42	43	44	45	46	47	48	49	7B	7B	7B	7B	7B	7B

ZK-4251-85

## Packed Decimal to Unsigned Trailing Numeric Value Translation Table

- The number on the left represents the low-order bits of the packed decimal value in hexadecimal notation.
- The number across the top represents the high-order bits of the packed decimal value in hexadecimal notation.
- The number in the body of the table represents the equivalent unsigned trailing numeric value in hexadecimal notation.

Table LIB-12 LIB\$AB\_CVTPT\_U

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A	30	31	32	33	34	35	36	37	38	39	00	00	00	00	00	00
B	30	31	32	33	34	35	36	37	38	39	00	00	00	00	00	00
C	30	31	32	33	34	35	36	37	38	39	00	00	00	00	00	00
D	30	31	32	33	34	35	36	37	38	39	00	00	00	00	00	00
E	30	31	32	33	34	35	36	37	38	39	00	00	00	00	00	00
F	30	31	32	33	34	35	36	37	38	39	00	00	00	00	00	00

ZK-4252-85

### Trailing Overpunch Numeric to Packed Decimal Value Translation Table

- The number on the left represents the low-order bits of the trailing overpunch numeric value in hexadecimal notation.
- The number across the top represents the high-order bits of the trailing overpunch numeric value in hexadecimal notation.
- The number in the body of the table represents the equivalent packed decimal value in hexadecimal notation.



# LIB\$MOVTC

**Table LIB-13 LIB\$AB\_CVTP\_O**

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	00	00	0C	00	7D	00	00	00	00	00	00	00	00	00	00
1	00	00	0D	1C	1C	8D	00	00	00	00	00	00	00	00	00	00
2	00	00	00	2C	2C	9D	00	00	00	00	00	00	00	00	00	00
3	00	00	00	3C	3C	00	00	00	00	00	00	00	00	00	00	00
4	00	00	00	4C	4C	00	00	00	00	00	00	00	00	00	00	00
5	00	00	00	5C	5C	00	00	00	00	00	00	00	00	00	00	00
6	00	00	00	6C	6C	00	00	00	00	00	00	00	00	00	00	00
7	00	00	00	7C	7C	00	00	00	00	00	00	00	00	00	00	00
8	00	00	00	8C	8C	00	00	00	00	00	00	00	00	00	00	00
9	00	00	00	9C	9C	00	00	00	00	00	00	00	00	00	00	00
A	00	00	00	0D	1D	00	00	00	00	00	00	00	00	00	00	00
B	00	00	00	00	2D	0C	00	0C	00	00	00	00	00	00	00	00
C	00	00	00	00	3D	00	00	00	00	00	00	00	00	00	00	00
D	00	00	00	00	4D	0D	00	0D	00	00	00	00	00	00	00	00
E	00	00	00	00	5D	00	00	00	00	00	00	00	00	00	00	00
F	00	00	00	0C	6D	00	00	00	00	00	00	00	00	00	00	00

ZK-4253-85

## Unsigned Numeric to Packed Decimal Value Translation Table

- The number on the left represents the low-order bits of the unsigned numeric value in hexadecimal notation.
- The number across the top represents the high-order bits of the unsigned numeric value in hexadecimal notation.
- The number in the body of the table represents the equivalent packed decimal value in hexadecimal notation.

Table LIB-14 LIB\$AB\_CVTTP\_U

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	00	00	0C	00	00	00	00	00	00	00	00	00	00	00	00
1	00	00	00	1C	00	00	00	00	00	00	00	00	00	00	00	00
2	00	00	00	2C	00	00	00	00	00	00	00	00	00	00	00	00
3	00	00	00	3C	00	00	00	00	00	00	00	00	00	00	00	00
4	00	00	00	4C	00	00	00	00	00	00	00	00	00	00	00	00
5	00	00	00	5C	00	00	00	00	00	00	00	00	00	00	00	00
6	00	00	00	6C	00	00	00	00	00	00	00	00	00	00	00	00
7	00	00	00	7C	00	00	00	00	00	00	00	00	00	00	00	00
8	00	00	00	8C	00	00	00	00	00	00	00	00	00	00	00	00
9	00	00	00	9C	00	00	00	00	00	00	00	00	00	00	00	00
A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

ZK-4254-85

### Trailing Overpunch Numeric to Unsigned Numeric Value Translation Table

- The number on the left represents the low-order bits of the trailing overpunch numeric value in hexadecimal notation.
- The number across the top represents the high-order bits of the trailing overpunch numeric value in hexadecimal notation.
- The number in the body of the table represents the equivalent unsigned numeric value in hexadecimal notation.

# LIB\$MOVTC

**Table LIB-15 LIB\$AB\_CVT\_O\_U**

Row bits 0 - 3	Column bits 4 - 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	20	30	40	37	60	70	80	90	A0	B0	C0	D0	E0	F0
1	01	11	30	31	31	38	61	71	81	91	A1	B1	C1	D1	E1	F1
2	02	12	22	32	32	39	62	72	82	92	A2	B2	C2	D2	E2	F2
3	03	13	23	33	33	53	63	73	83	93	A3	B3	C3	D3	E3	F3
4	04	14	24	34	34	54	64	74	84	94	A4	B4	C4	D4	E4	F4
5	05	15	25	35	35	55	65	75	85	95	A5	B5	C5	D5	E5	F5
6	06	16	26	36	36	56	66	76	86	96	A6	B6	C6	D6	E6	F6
7	07	17	27	37	37	57	67	77	87	97	A7	B7	C7	D7	E7	F7
8	08	18	28	38	38	58	68	78	88	98	A8	B8	C8	D8	E8	F8
9	09	19	29	39	39	59	69	79	89	99	A9	B9	C9	D9	E9	F9
A	0A	1A	2A	30	31	5A	6A	7A	8A	9A	AA	BA	CA	DA	EA	FA
B	0B	1B	2B	3B	32	30	6B	30	8B	9B	AB	BB	CB	DB	EB	FB
C	0C	1C	2C	3C	33	5C	6C	7C	8C	9C	AC	BC	CC	DC	EC	FC
D	0D	1D	2D	3D	34	30	6D	30	8D	9D	AD	BD	CD	DD	ED	FD
E	0E	1E	2E	3E	35	5E	6E	7E	8E	9E	AE	BE	CE	DE	EE	FE
F	0F	1F	2F	30	36	5F	6F	7F	8F	9F	AF	BF	CF	DF	EF	FF

ZK-4255-85

## Unsigned Numeric to Trailing Overpunch Translation Table

Table LIB-16 is indexed by 0 through 9 for the positive overpunches and 10 through 19 for the negative overpunches.

The unsigned binary representation of the least significant digit is moved into R2. Then, if you require a positive result, the following code results:

```
MOV C3 LIB$AB_CVT_U_0[R2], #1,R0
```

If you require a negative result, the following code is generated:

```
MOV C3 LIB$AV_CVT_U_0 + 10[R2], #1,R0
```

The result is the overpunch representation for the last byte of the negative number.

**Table LIB-16 LIB\$AB\_CVT\_U\_O**

0 - 9	10 - 19
7B 41 42 43 44 45 46 47 48 49	7D 4A 4B 4C 4D 4E 4F 50 51 52

ZK-4256-85

Table LIB-17 LIB\$AB\_CVTPT\_Z

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
1	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
2	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
3	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
4	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
5	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
6	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
7	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
8	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
9	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
A	30	31	32	33	34	35	36	37	38	39	30	30	30	30	30	30
B	70	71	72	73	74	75	76	77	78	79	30	30	30	30	30	30
C	30	31	32	33	34	35	36	37	38	39	30	30	30	30	30	30
D	70	71	72	73	74	75	76	77	78	79	30	30	30	30	30	30
E	30	31	32	33	34	35	36	37	38	39	30	30	30	30	30	30
F	30	31	32	33	34	35	36	37	38	39	30	30	30	30	30	30

ZK-6414-HC

**Packed Decimal to Zone Numeric Translation Table**

- The number on the left represents the low-order bits of the packed decimal value in hexadecimal notation.
- The number across the top represents the high-order bits of the packed decimal value in hexadecimal notation.
- The number in the body of the table represents the equivalent zoned numeric value in hexadecimal notation.

# LIB\$MOVTC

**Table LIB-18 LIB\$AB\_CVTP\_Z**

Row bits 0 - 3	Column bits 4 - 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	00	00	0C	00	00	00	0D	00	00	00	00	00	00	00	00
1	00	00	00	1C	00	00	00	1D	00	00	00	00	00	00	00	00
2	00	00	00	2C	00	00	00	2D	00	00	00	00	00	00	00	00
3	00	00	00	3C	00	00	00	3D	00	00	00	00	00	00	00	00
4	00	00	00	4C	00	00	00	4D	00	00	00	00	00	00	00	00
5	00	00	00	5C	00	00	00	5D	00	00	00	00	00	00	00	00
6	00	00	00	6C	00	00	00	6D	00	00	00	00	00	00	00	00
7	00	00	00	7C	00	00	00	7D	00	00	00	00	00	00	00	00
8	00	00	00	8C	00	00	00	8D	00	00	00	00	00	00	00	00
9	00	00	00	9C	00	00	00	9D	00	00	00	00	00	00	00	00
A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

ZK-6415-HC

## Zone to Packed Decimal Translation Table

- The number on the left represents the low-order bits of the zoned numeric value in hexadecimal notation.
- The number across the top represents the high-order bits of the zoned numeric value in hexadecimal notation.
- The number in the body of the table represents the equivalent packed value decimal value in hexadecimal notation.

Table LIB-19 LIB\$AB\_UPCASE

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	20	30	40	50	60	50	80	90	A0	B0	C0	D0	C0	F0
1	01	11	21	31	41	51	41	51	81	91	A1	B1	C1	D1	C1	F1
2	02	12	22	32	42	52	42	52	82	92	A2	B2	C2	D2	C2	F2
3	03	13	23	33	43	53	43	53	83	93	A3	B3	C3	D3	C3	F3
4	04	14	24	34	44	54	44	54	84	94	A4	B4	C4	D4	C4	F4
5	05	15	25	35	45	55	45	55	85	95	A5	B5	C5	D5	C5	F5
6	06	16	26	36	46	56	46	56	86	96	A6	B6	C6	D6	C6	F6
7	07	17	27	37	47	57	47	57	87	97	A7	B7	C7	D7	C7	F7
8	08	18	28	38	48	58	48	58	88	98	A8	B8	C8	D8	C8	F8
9	09	19	29	39	49	59	49	59	89	99	A9	B9	C9	D9	C9	F9
A	0A	1A	2A	3A	4A	5A	4A	5A	8A	9A	AA	BA	CA	DA	CA	DA
B	0B	1B	2B	3B	4B	5B	4B	5B	8B	9B	AB	BB	CB	DB	CB	DB
C	0C	1C	2C	3C	4C	5C	4C	5C	8C	9C	AC	BC	CC	DC	CC	DC
D	0D	1D	2D	3D	4D	5D	4D	5D	8D	9D	AD	BD	CD	DD	CD	DD
E	0E	1E	2E	3E	4E	5E	4E	5E	8E	9E	AE	BE	CE	DE	CE	FE
F	0F	1F	2F	3F	4F	5F	4F	5F	8F	9F	AF	BF	CF	DF	CF	FF

ZK-6416-HC

**ASCII Uppercase Translation Table**

- The number on the left represents the low-order bits of the ASCII character in hexadecimal notation.
- The number across the top represents the high-order bits of the ASCII character in hexadecimal notation.
- The number in the body of the table represents the equivalent uppercase ASCII character in hexadecimal notation.

# LIB\$MOVTC

**Table LIB-20 LIB\$AB\_LOWERCASE**

Row bits 0 - 3	Column bits 4 - 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	20	30	40	70	60	70	80	90	A0	B0	E0	D0	E0	F0
1	01	11	21	31	61	71	61	71	81	91	A1	B1	E1	F1	E1	F1
2	02	12	22	32	62	72	62	72	82	92	A2	B2	E2	F2	E2	F2
3	03	13	23	33	63	73	63	73	83	93	A3	B3	E3	F3	E3	F3
4	04	14	24	34	64	74	64	74	84	94	A4	B4	E4	F4	E4	F4
5	05	15	25	35	65	75	65	75	85	95	A5	B5	E5	F5	E5	F5
6	06	16	26	36	66	76	66	76	86	96	A6	B6	E6	F6	E6	F6
7	07	17	27	37	67	77	67	77	87	97	A7	B7	E7	F7	E7	F7
8	08	18	28	38	68	78	68	78	88	98	A8	B8	E8	F8	E8	F8
9	09	19	29	39	69	79	69	79	89	99	A9	B9	E9	F9	E9	F9
A	0A	1A	2A	3A	6A	7A	6A	7A	8A	9A	AA	BA	EA	FA	EA	FA
B	0B	1B	2B	3B	6B	7B	6B	7B	8B	9B	AB	BB	EB	FB	EB	FB
C	0C	1C	2C	3C	6C	7C	6C	7C	8C	9C	AC	BC	EC	FC	EC	FC
D	0D	1D	2D	3D	6D	7D	6D	7D	8D	9D	AD	BD	ED	FD	ED	FD
E	0E	1E	2E	3E	6E	7E	6E	7E	8E	9E	AE	BE	EE	FE	EE	FE
F	0F	1F	2F	3F	6F	7F	6F	7F	8F	9F	AF	BF	EF	FF	EF	FF

ZK-6417-HC

## ASCII Lowercase Translation Table

- The number on the left represents the low-order bits of the ASCII character in hexadecimal notation.
- The number across the top represents the high-order bits of the ASCII character in hexadecimal notation.
- The number in the body of the table represents the equivalent lowercase ASCII character in hexadecimal notation.

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Routine successfully completed; string truncated. The fixed-length destination string could not contain all the characters.
LIB\$_FATERRLIB	Fatal internal error.
LIB\$_INSVIRMEM	Insufficient virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor.

---

**EXAMPLE**

```

1  !+
   !This BASIC program illustrates the method
   !of creating a descriptor for the appropriate
   !translation table in order to call LIB$MOVTC.
   !-

   OPTION TYPE = EXPLICIT

   !+
   !Declare the translation table as an
   !EXTERNAL LONG variable.
   !-

   EXTERNAL LONG LIB$AB_ASC_EBC
   EXTERNAL LONG FUNCTION LIB$MOVTC
   EXTERNAL SUB LIB$STOP
   EXTERNAL LONG CONSTANT DSC$K_CLASS_S, DSC$K_DTYPE_T

   !+
   !Define a record which models the required
   !translation table descriptor.
   !-

   RECORD STR_TYPE
       BYTE    DSC$B_CLASS
       BYTE    DSC$B_DTYPE
       WORD    DSC$W_LENGTH
       LONG    DSC$A_POINTER
   END RECORD STR_TYPE

   DECLARE LONG I, RET_STS
   DECLARE STR_TYPE STR_VAR

   MAP (FOO) STRING DST = 3%
   MAP (FOO) BYTE DST_ARRAY(2)

   !+
   !Fill the translation table descriptor record.
   !Note that the length of the translation table string
   !is set to 256, and the pointer receives the address of
   !the DIGITAL translation table LIB$AB_ASC_EBC.
   !-

   STR_VAR::DSC$B_CLASS = DSC$K_CLASS_S
   STR_VAR::DSC$B_DTYPE = DSC$K_DTYPE_T
   STR_VAR::DSC$W_LENGTH = 256
   STR_VAR::DSC$A_POINTER = LOC(LIB$AB_ASC_EBC)

   RET_STS = LIB$MOVTC( "ABC", " ", STR_VAR BY REF, DST )
   IF (RET_STS AND 1%) = 0%
   THEN
       CALL LIB$STOP( RET_STS BY VALUE )
   END IF

   !+
   !Add 256 to the translated value in order to return
   !an unsigned value.
   !-

```



# LIB\$MOVTC

```
PRINT (256 + DST_ARRAY(I)) FOR I = 0% TO 2%
```

```
END
```

The output generated by this program is as follows:

```
193  
194  
195
```



# LIB\$MOVTUC

You can use any of the translation tables included in the Description section of LIB\$MOVTC, or you can create your own. When using a translation table supplied by DIGITAL, the names LIB\$AB\_XXX\_YYY represent the addresses of the 256 byte translation tables, and can be accessed as external (string) variables. If a particular language cannot generate descriptors for external strings, then they must be created manually. The example for the routine LIB\$MOVTC illustrates the creation of a string descriptor for a translation table using VAX BASIC.

## ***destination-string***

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

Destination string into which LIB\$MOVTUC writes the translated **source-string**. The **destination-string** argument is the address of a descriptor pointing to this destination string.

## ***fill-character***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Character used to pad **source-string** to the length of **destination-string**. The **fill-character** argument is the address of a descriptor pointing to a string. The first character of this string is used as the fill character. The length of this string is not checked and **fill-character** is not translated.

If the fill character is included, the remainder of the destination string (after the stop character) is filled with the specified fill character. If it is not included, the remainder of the destination string remains unchanged.

---

## **DESCRIPTION**

During the translation, LIB\$MOVTUC accesses each character in the source string and uses it as an index into the translation table. If the table entry contains the specified stop character, the routine is terminated and the relative position of the source character is returned.

If the source string is longer than the destination string, then the source string is truncated. If the optional fill character is present, any remaining positions in the destination string are filled with the fill character. If the source or destination string is exhausted (before the stop character is found), a zero index is returned.

The results are unpredictable if the source and destination strings overlap and have different starting addresses.

See the description of LIB\$MOVTC for the translation tables used by LIB\$MOVTC and LIB\$MOVTUC. Each translation table is preceded by explanatory text.

---

## **CONDITION VALUES RETURNED**

*None.*

---

## LIB\$MULT\_DELTA\_TIME    Multiply Delta Time by Scalar

The Multiply Delta Time by Scalar routine multiplies a delta time by a longword integer scalar.

---

**FORMAT**                    **LIB\$MULT\_DELTA\_TIME**    *multiplier ,delta-time*

---

**RETURNS**                    VMS usage: **cond\_value**  
                                   type:            **longword (unsigned)**  
                                   access:        **write only**  
                                   mechanism:    **by value**

---

**ARGUMENTS**                ***multiplier***  
                                   VMS usage: **longword\_signed**  
                                   type:            **longword (signed)**  
                                   access:        **read only**  
                                   mechanism:    **by reference**

The value by which LIB\$MULT\_DELTA\_TIME multiplies the delta time. The **multiplier** argument is the address of a signed longword containing the integer scalar. If **multiplier** is negative, the absolute value of **multiplier** is used.

***delta-time***  
 VMS usage: **date\_time**  
 type:            **quadword (unsigned)**  
 access:        **modify**  
 mechanism:    **by reference**

The delta time to be multiplied. The **delta-time** argument is the address of an unsigned quadword containing the number to be multiplied. **Delta-time** must be less than 10,000 days. After LIB\$MULT\_DELTA\_TIME performs the multiplication, the result is returned to **delta-time**. (The original **delta-time** is overwritten.)

---

**DESCRIPTION**                LIB\$MULT\_DELTA\_TIME multiplies a delta time by a longword integer scalar. The result of the multiplication is returned to the **delta-time** argument.

---

<b>CONDITION VALUES RETURNED</b>	LIB\$_NORMAL	Normal successful completion.
	LIB\$_IVTIME	Invalid time.
	LIB\$_WRONUMARG	Incorrect number of arguments.

# LIB\$MULTF\_DELTA\_TIME

---

## LIB\$MULTF\_DELTA\_TIME Multiply Delta Time by an F\_Floating Scalar

The Multiply Delta Time by an F-Floating Scalar routine multiplies a delta time by an F-floating scalar.

---

**FORMAT**            **LIB\$MULTF\_DELTA\_TIME**    *multiplier ,delta-time*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:         **write only**  
                          mechanism:      **by value**

---

**ARGUMENTS**         ***multiplier***  
                          VMS usage: **floating\_point**  
                          type:            **F\_floating**  
                          access:         **read only**  
                          mechanism:      **by reference**

The value by which LIB\$MULTF\_DELTA\_TIME multiplies the delta time. The **multiplier** argument is the address of an F-floating value containing the scalar. If **multiplier** is negative, the absolute value of **multiplier** is used.

***delta-time***  
VMS usage: **date\_time**  
type:            **quadword (unsigned)**  
access:         **modify**  
mechanism:      **by reference**

The delta time to be multiplied. The **delta-time** argument is the address of an unsigned quadword containing the number to be multiplied. **Delta-time** must be less than 10,000 days. After LIB\$MULTF\_DELTA\_TIME performs the multiplication, the result is returned to **delta-time**. (The original **delta-time** is overwritten.)

---

**DESCRIPTION**        LIB\$MULTF\_DELTA\_TIME multiplies a delta time by an F-floating scalar. The result of the multiplication is returned to the **delta-time** argument.

---

**CONDITION VALUES RETURNED**

LIB\$_NORMAL	Normal successful completion.
LIB\$_IVTIME	Invalid time.
LIB\$_WRONUMARG	Incorrect number of arguments.

---

## LIB\$PAUSE Pause Program Execution

The Pause Program Execution routine suspends program execution and returns control to the calling command level.

---

**FORMAT**                    **LIB\$PAUSE**

---

**RETURNS**                    VMS usage: **cond\_value**  
                                   type:            **longword (unsigned)**  
                                   access:        **write only**  
                                   mechanism:    **by value**

---

**ARGUMENTS**                *None.*

---

**DESCRIPTION**              LIB\$PAUSE suspends program execution and returns control to the calling command level. The suspended image may be continued with the CONTINUE command, or it may be terminated with the EXIT or STOP commands. In the latter case, the image will not return to this routine.

Note that this routine functions only for interactive jobs. If this routine is invoked in batch mode, it has no effect.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Normal successful completion.
LIB\$_NOCLI	No CLI present. The calling process does not have a CLI or the CLI does not support the request. Note that DCL only supports this function in INTERACTIVE mode.

# LIB\$POLYD

---

## LIB\$POLYD Evaluate Polynomials

The Evaluate Polynomials routine (D-floating point values) allows higher-level language users to evaluate D-floating point value polynomials.

---

**FORMAT**            **LIB\$POLYD** *polynomial-argument ,degree ,coefficient ,floating-point-result*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:        **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        ***polynomial-argument***  
                          VMS usage: **floating\_point**  
                          type:        **D\_floating**  
                          access:     **read only**  
                          mechanism: **by reference**

Argument for the polynomial. The **polynomial-argument** argument is the address of a floating-point number that contains this argument. The **polynomial-argument** argument is a D-floating number.

### ***degree***

VMS usage: **word\_signed**  
type:        **word integer (signed)**  
access:     **read only**  
mechanism: **by reference**

Highest numbered nonzero coefficient to participate in the evaluation. The **degree** argument is the address of a signed word integer that contains this highest-numbered coefficient.

If the degree is 0, the result equals C[0]. The range of the degree is 0 to 31.

### ***coefficient***

VMS usage: **floating\_point**  
type:        **D\_floating**  
access:     **read only**  
mechanism: **by reference, array reference**

Floating-point coefficients. The **coefficient** argument is the address of an array of floating-point coefficients. The coefficient of the highest-order term of the polynomial is the lowest-addressed element in the array. The **coefficient** argument is an array of D-floating numbers.

***floating-point-result***

VMS usage: **floating\_point**  
 type: **D\_floating**  
 access: **write only**  
 mechanism: **by reference**

Result of the calculation. The **floating-point-result** argument is the address of a floating-point number that contains this result. LIB\$POLYD writes the address of **floating-point-result** into a D-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (63 bits for POLYD).

---

**DESCRIPTION**

LIB\$POLYD provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner's Method. The result is computed as follows:

$$\text{result} = C[0] + X * (C[1] + X * (C[2] + \dots X * (C[D]) \dots))$$

In the above result *D* is the degree of the polynomial and *X* is the argument.

See the *VAX Architecture Reference Manual* for the detailed description of POLY.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
SS\$_FLTOVF	Floating overflow.
SS\$_ROPRAND	Reserved operand.

---

**EXAMPLE**

The FORTRAN and Pascal examples provided in the description of LIB\$POLYF also demonstrate how to use LIB\$POLYD. Please refer to those examples for assistance in using this routine.





**floating-point-result**

VMS usage: **floating\_point**  
 type: **F\_floating**  
 access: **write only**  
 mechanism: **by reference**

Result of the calculation. The **floating-point-result** argument is the address of a floating-point number that contains this result. LIB\$POLYF writes the address of **floating-point-result** into an F-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (31 bits for POLYF).

**DESCRIPTION**

LIB\$POLYF provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner's Method. The result is computed as follows:

$$\text{result} = C[0] + X * (C[1] + X * (C[2] + \dots X * (C[D]) \dots))$$

In the above result  $D$  is the degree of the polynomial and  $X$  is the argument.

**CONDITION VALUES RETURNED**

SS\$_NORMAL	Normal successful completion.
SS\$_FLTOVF	Floating overflow.
SS\$_ROPRAND	Reserved operand.

**EXAMPLES****1**

```
C+
C This FORTRAN example demonstrates how to use
C LIB$POLYF.
C-
```

```
REAL*4 X,COEFF(5),RESULT
INTEGER*2 DEG
```

```
C+
C Compute X^4 + 2*X^3 -X^2 + X - 3 using POLYF.
C Let X = 2.
C The coefficients needed are as follows:
C-
```

```
DATA COEFF/1.0,2.0,-1.0,1.0,-3.0/
X = 2.0
DEG = 4
```

```
! DEG has word length.
```

```
C+
C Calculate (2)^4 + 2*(2^3) -2^2 + 2 - 3.
C The result should be 27.
C-
```

```
RETURN = LIB$POLYF(X,DEG,COEFF,RESULT)
TYPE *, '(2)^4 + 2*(2^3) -2^2 + 2 - 3 = ',RESULT
END
```

# LIB\$POLYF

This FORTRAN example demonstrates how to call LIB\$POLYF. The output generated by this program is as follows:

$$(2)^4 + 2*(2^3) - 2^2 + 2 - 3 = 27.00000$$

2

```
PROGRAM POLYF(INPUT,OUTPUT);
{+}
{ This Pascal program demonstrates how to use
{ LIB$POLYF to evaluate a polynomial.
{-}

TYPE
  WORD = [WORD] 0..65535;

VAR
  COEFF : ARRAY [0..2] OF REAL := (1.0,2.0,2.0);
  RESULT : REAL;
  RETURNED_STATUS : INTEGER;

[EXTERNAL] FUNCTION LIB$POLYF(
  ARG : REAL;
  DEGREE : WORD;
  COEFF : [REFERENCE] ARRAY [L..U:INTEGER] OF REAL;
  VAR RESULT : REAL
  ) : INTEGER; EXTERNAL;

[EXTERNAL] FUNCTION LIB$STOP(
  CONDITION_STATUS : [IMMEDIATE,UNSAFE] UNSIGNED;
  FAO_ARGS : [IMMEDIATE,UNSAFE,LIST] UNSIGNED
  ) : INTEGER; EXTERNAL;

BEGIN
{+}
{ Call LIB$POLYF to evaluate 2(X**2) + 2*X + 1.
{-}

RETURNED_STATUS := LIB$POLYF(1.0,2,COEFF,RESULT);
IF NOT ODD(RETURNED_STATUS)
THEN
  LIB$STOP(RETURNED_STATUS);

WRITELN('F(1.0) = ',RESULT:5:2);

END.
```

This example program demonstrates how to call LIB\$POLYF from Pascal. The output generated by this Pascal program is as follows:

```
$ RUN POLYF
F(1.0) = 5.00
```



# LIB\$POLYG

## *floating-point-result*

VMS usage: **floating\_point**  
type: **G\_floating**  
access: **write only**  
mechanism: **by reference**

Result of the calculation. The **floating-point-result** argument is the address of a floating-point number that contains this result. LIB\$POLYG writes the address of **floating-point-result** into a G-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (63 bits for POLYG).

---

## DESCRIPTION

LIB\$POLYG provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner's Method. The result is computed as follows:

$$\text{result} = C[0] + X * (C[1] + X * (C[2] + \dots X * (C[D]) \dots))$$

In the above result *D* is the degree of the polynomial and *X* is the argument.

See the *VAX Architecture Reference Manual* for the detailed description of POLY.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
SS\$_FLTOVF	Floating overflow.
SS\$_ROPRAND	Reserved operand.

---

## EXAMPLE

The FORTRAN and Pascal examples provided in the description of LIB\$POLYF also demonstrate how to use LIB\$POLYG. Please refer to those examples for assistance in using this routine.



# LIB\$POLYH

## ***floating-point-result***

VMS usage: **floating\_point**  
type: **H\_floating**  
access: **write only**  
mechanism: **by reference**

Result of the calculation. The **floating-point-result** argument is the address of a floating-point number that contains this result. LIB\$POLYH writes the address of **floating-point-result** into an H-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (127 bits for POLYH).

---

## **DESCRIPTION**

LIB\$POLYH provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner's Method. The result is computed as follows:

$result = C[0] + X * (C[1] + X * (C[2] + \dots X * (C[D]) \dots))$

In the above result *D* is the degree of the polynomial and *X* is the argument.

See the *VAX Architecture Reference Manual* for the detailed description of POLY.

---

## **CONDITION VALUES RETURNED**

SS\$_NORMAL	Routine successfully completed.
SS\$_FLTUVF	Floating overflow.
SS\$_ROPRAND	Reserved operand.

---

## **EXAMPLE**

The FORTRAN and Pascal examples provided in the description of LIB\$POLYF also demonstrate how to use LIB\$POLYH. Please refer to those examples for assistance in using this routine.

---

## LIB\$PUT\_COMMON Put String to Common

The Put String to Common routine copies the contents of a string into the common area. The common area is an area of storage which remains defined across multiple image activations in a process. Optionally, LIB\$PUT\_COMMON returns the actual number of characters copied. The maximum number of characters that can be copied is 252.

---

**FORMAT**            **LIB\$PUT\_COMMON**    *source-string [,resultant-length]*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        ***source-string***  
                           VMS usage: **char\_string**  
                           type:        **character string**  
                           access:     **read only**  
                           mechanism: **by descriptor**

Source string to be copied to the common area by LIB\$PUT\_COMMON. The **source-string** argument is the address of a descriptor pointing to this source string.

***resultant-length***  
 VMS usage: **word\_unsigned**  
 type:        **word (unsigned)**  
 access:     **write only**  
 mechanism: **by reference**

Number of characters copied by LIB\$PUT\_COMMON to the common area. The **resultant-length** argument is the address of an unsigned word integer that contains this number of characters. LIB\$PUT\_COMMON writes this number into the **resultant-length** argument.

---

**DESCRIPTION**      LIB\$PUT\_COMMON and LIB\$GET\_COMMON allow programs to copy strings to and from the common area. The programs reading and writing the data in the common area must agree upon its amount and format. The maximum length of the destination string is defined as follows:

[min(256, the length of the data in the common storage area) - 4]

Thus, maximum length is 252.

In BASIC and FORTRAN, you can use these routines to allow a USEROPEN routine to pass information back to the routine that called it. A USEROPEN routine cannot write arguments. However, it can call LIB\$PUT\_COMMON to put information into the common area. The calling program can then use LIB\$GET\_COMMON to retrieve it.



# LIB\$PUT\_COMMON

You can also use these routines to pass information between images run successively, such as chained images run by LIB\$RUN\_PROGRAM. Since the common area is unique to each process, do not use LIB\$GET\_COMMON and LIB\$PUT\_COMMON to share information across processes.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Successfully completed, but the source string was truncated.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.

---

## LIB\$PUT\_OUTPUT Put Line to SYSS\$OUTPUT

The Put Line to SYSS\$OUTPUT routine writes a record to the current controlling output device, specified by SYSS\$OUTPUT using the RMS \$PUT service.

---

**FORMAT**            **LIB\$PUT\_OUTPUT**    *message-string*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENT**            ***message-string***  
                           VMS usage: **char\_string**  
                           type:        **character string**  
                           access:     **read only**  
                           mechanism: **by descriptor**

Message string written to the current controlling output device by LIB\$PUT\_OUTPUT. The **message-string** argument is the address of a descriptor pointing to this message string. RMS handles all formatting, so the message does not need to include such ASCII formatting instructions as carriage return (CR).

---

**DESCRIPTION**        When you log in, VMS creates three files as default I/O control streams for your process.

- SYSS\$INPUT, your default input device
- SYSS\$OUTPUT, your default output device
- SYSS\$COMMAND, the device that supplies the commands to your process

These files remain open until you log out. They are the interface between your interactive input and output or batch commands and the VMS software. Initially, all three are equated with the terminal. However, with the DCL ASSIGN command, you can change these assignments to obtain information from a file or put information into a file. SYSS\$INPUT and SYSS\$COMMAND are usually identical, but the input and command streams can be different. For example, during the execution of an indirect command file from an interactive terminal, SYSS\$COMMAND refers to the terminal and SYSS\$INPUT refers to the command file.

On the first call to LIB\$PUT\_OUTPUT, if the output file is not a process-permanent file, LIB\$PUT\_OUTPUT opens the output file and positions it at the end-of-file mark. If no output file exists on the first call, LIB\$PUT\_OUTPUT creates a file. The RMS internal stream identifier (ISI) is stored in the routine's static storage for subsequent calls. Hence, this routine is not AST reentrant.

# LIB\$PUT\_OUTPUT

LIB\$PUT\_OUTPUT uses RMS to format records on output, and RMS records have implied carriage control. That is, a record normally corresponds to a line of text. Therefore, if you want explicit carriage control, instead of implied carriage control, you must supply it yourself within the source string.

LIB\$PUT\_OUTPUT is the most convenient way for a MACRO or BLISS program to write information to SYS\$OUTPUT.

If you have several shareable images that call LIB\$PUT\_OUTPUT, and if each shareable image includes its own copy of LIB\$PUT\_OUTPUT, your program could produce multiple output streams and multiple versions of your output file. A single application should reference one copy of LIB\$PUT\_OUTPUT.

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Routine successfully completed.

Any condition values returned by RMS.

---

## EXAMPLE

```
10      !+
      ! This BASIC program demonstrates how to use
      ! LIB$PUT_OUTPUT to output a simple message.
      !-
      MSGSTR$ = 'This is a sample message'
      CALL LIB$PUT_OUTPUT(MSGSTR$)
      !+
      ! In this example, the default value of
      ! SYS$OUTPUT is used. Therefore, the
      ! output is 'put' to the terminal screen.
      !-
90      END
```

This BASIC program illustrates the use of LIB\$PUT\_OUTPUT. The output generated by this BASIC example is as follows:

This is a sample message

---

## LIB\$RADIX\_POINT Radix Point Symbol

The Radix Point Symbol routine returns the system's radix point symbol. This symbol is used inside a digit string to separate the integer part from the fraction part. This routine works by attempting to translate the logical name SYS\$RADIX\_POINT as a process, group, or system logical name.

---

**FORMAT**            **LIB\$RADIX\_POINT**    *radix-point-string* [, *resultant-length*]

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        ***radix-point-string***  
                           VMS usage: **char\_string**  
                           type:        **character string**  
                           access:      **write only**  
                           mechanism: **by descriptor**

Radix point string. The **radix-point-string** argument is the address of a descriptor pointing to this radix point string.

***resultant-length***  
 VMS usage: **word\_unsigned**  
 type:        **word (unsigned)**  
 access:      **write only**  
 mechanism: **by reference**

The number of characters written into **radix-point-string**, not counting padding in the case of a fixed-length string. The **resultant-length** argument is the address of an unsigned word that contains this number.

If the **radix-point-string** argument is the address of a fixed-length string descriptor, there may not be enough characters in the fixed-length string to contain the whole radix point string, and the radix point string is truncated. If the radix point string is truncated to the size specified in a fixed-length string descriptor, **resultant-length** is set to this size. Therefore, **resultant-length** can always be used by the calling program to access a valid substring of **radix-point-string**.

---

**DESCRIPTION**        If unable to translate the logical name SYS\$RADIX\_POINT, LIB\$RADIX\_POINT returns the United States radix point symbol (.). If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS\$RADIX\_POINT as a system-wide logical name to provide a default for all users, and an individual user with a special need can define SYS\$RADIX\_POINT as a process logical name to override the default.

LIB\$RADIX\_POINT is used implicitly by BASIC.

# LIB\$RADIX\_POINT

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Normal successful completion.

LIB\$\_STRTRU

Successfully completed, but the radix point string was truncated.

LIB\$\_FATERRLIB

Fatal internal error.

LIB\$\_INSVIRMEM

Insufficient virtual memory.

LIB\$\_INVSTRDES

Invalid string descriptor.

---

## LIB\$REMQHI Remove Entry from Head of Queue

The Remove Entry from Head of Queue routine removes an entry from the head of the specified self-relative interlocked queue. LIB\$REMQHI makes the VAX REMQHI instruction available as a callable routine.

---

**FORMAT**            **LIB\$REMQHI** *header ,remque-address [,retry-count]*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:          **write only**  
                           mechanism:       **by value**

---

### ARGUMENTS

#### *header*

VMS usage: **quadword\_signed**  
 type:            **quadword integer (signed)**  
 access:          **modify**  
 mechanism:       **by reference**

Queue header specifying the queue from which **entry** will be removed. The **header** argument contains the address of this signed aligned quadword integer. **Header** must be initialized to zero before first use of the queue; zero means an empty queue.

#### *remque-address*

VMS usage: **address**  
 type:            **longword (unsigned)**  
 access:          **write only**  
 mechanism:       **by reference**

Address of the removed entry. The **remque-address** argument is the address of an unsigned longword that contains this address. If the queue was empty, **remque-address** is set to the address of the header.

#### *retry-count*

VMS usage: **longword\_unsigned**  
 type:            **longword (unsigned)**  
 access:          **read only**  
 mechanism:       **by reference**

The number of times the operation is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The **retry-count** argument is the address of a longword that contains the retry count value. A value of 1 causes no retries. The default value is 10.

# LIB\$REMQHI

---

## DESCRIPTION

The queue from which LIB\$REMQHI removes an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address. Two longwords, a forward link and a backward link, define the location of the entry in relation to the preceding and succeeding entries.

A self-relative queue is a queue in which the links between entries are displacements; the two longwords represent the displacements of the current entry's predecessor and successor. The VAX instructions INSQHI and REMQHI allow you to insert and remove an entry at the head of a self-relative queue. The corresponding Run-Time Library routines are LIB\$INSQHI and LIB\$REMQHI.

The self-relative queue instructions are interlocked and cannot be interrupted, so that other processes cannot insert or remove queue entries while the current program is doing so. Since the operation requires changing two pointers at the same time, a high-level language cannot perform this operation without calling the Run-Time Library queue access routines.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue Access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an asynchronous system trap.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. The entry was removed from the head of the queue, and the resulting queue contains one or more entries.
LIB\$_ONEENTQUE	Routine successfully completed. The entry was removed from the head of the queue, and the resulting queue is empty.
LIB\$_SECINTFAI	A secondary interlock failure occurred; the insertion was attempted the number of times specified by <b>retry-count</b> . This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.
LIB\$_QUEWASEMP	The queue was empty. The queue is not modified.
SS\$_ROPRAND	Reserved operand fault. Either the entry or the header is at an address that is not quadword aligned, or the header address equals the entry address.

---

## LIB\$REMQTI Remove Entry from Tail of Queue

The Remove Entry from Tail of Queue routine removes an entry from the tail of the specified self-relative interlocked queue. LIB\$REMQTI makes the VAX REMQTI instruction available as a callable routine.

---

**FORMAT**                    **LIB\$REMQTI** *header ,remque-address [,retry-count]*

---

**RETURNS**                    VMS usage: **cond\_value**  
                                   type:            **longword (unsigned)**  
                                   access:        **write only**  
                                   mechanism:    **by value**

---

**ARGUMENTS**                ***header***  
                                   VMS usage: **quadword\_signed**  
                                   type:        **quadword integer (signed)**  
                                   access:      **modify**  
                                   mechanism: **by reference**

Queue header specifying the queue from which the entry is to be deleted. The **header** argument contains the address of this signed aligned quadword integer. **Header** must be initialized to zero before first use of the queue; zero means an empty queue.

***remque-address***  
 VMS usage: **address**  
 type:        **longword (unsigned)**  
 access:     **write only**  
 mechanism: **by reference**

Address of the removed entry. The **remque-address** argument is the address of a longword that contains this address. If the queue was empty, **remque-address** is set to the address of the header.

***retry-count***  
 VMS usage: **longword\_unsigned**  
 type:        **longword (unsigned)**  
 access:     **read only**  
 mechanism: **by reference**

The number of times the operation is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The **retry-count** argument is the address of a longword that is this retry count value. A value of 1 causes no retries. The default value is 10.



# LIB\$REMQTI

---

## DESCRIPTION

The queue from which LIB\$REMQTI removes an in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address. Two longwords, a forward link and a backward link, define the location of the entry in relation to the preceding and succeeding entries.

A self-relative queue is a queue in which the links between entries are displacements; the two longwords represent the displacements of the current entry's predecessor and successor. The VAX instructions INSQTI and REMQTI allow you to insert and remove an entry at the tail of a self-relative queue. The corresponding Run-Time Library routines are LIB\$INSQTI and LIB\$REMQTI.

The self-relative queue instructions are interlocked and cannot be interrupted, so that other processes cannot insert or remove queue entries while the current program is doing so. Since the operation requires changing two pointers at the same time, a high-level language cannot perform this operation without calling the Run-Time Library queue access routines.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue Access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an asynchronous system trap.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. The entry was removed from the queue tail, and the resulting queue contains one or more entries.
SS\$_ROPRAND	Reserved operand fault. Either the entry or the header is at an address that is not quadword aligned, or the header address equals the entry address.
LIB\$_ONEENTQUE	Routine successfully completed. The entry was removed from the queue tail, and the resulting queue is empty.
LIB\$_QUEWASEMP	Queue was empty. The queue is not modified.
LIB\$_SECINTFAI	A secondary interlock failure occurred; the insertion was attempted the number of times specified by <b>retry-count</b> . This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.

---

## LIB\$RENAME\_FILE Rename One or More Files

The Rename One or More Files routine changes the names of one or more files. The specification of the files to be renamed may include wildcards.

LIB\$RENAME\_FILE is similar in function to the DCL command RENAME.

---

<b>FORMAT</b>	<b>LIB\$RENAME_FILE</b>	<i>old-filespec ,new-filespec</i> <i>[,default-filespec] [,related-filespec]</i> <i>[,flags] [,user-success-procedure]</i> <i>[,user-error-procedure]</i> <i>[,user-confirm-procedure]</i> <i>[,user-specified-argument]</i> <i>[,old-resultant-name]</i> <i>[,new-resultant-name]</i> <i>[,file-scan-context]</i>
---------------	-------------------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>old-filespec</i></b> VMS usage: <b>char_string</b> type: <b>character string</b> access: <b>read only</b> mechanism: <b>by descriptor</b>
------------------	---

File specification of the files to be renamed. The **old-filespec** argument is the address of a descriptor pointing to the old file specification. The specification may include wildcards, in which case each file which matches the specification will be renamed. The string must not contain more than 255 characters. Any string class is supported.

***new-filespec***  
VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

File specification for the new file names. The **new-filespec** argument is the address of a descriptor pointing to the new file specification.

This specification need not be complete; fields omitted or specified by using the wildcard character (\*) will be filled in from the existing file's name using the same rules as for the DCL command RENAME. The string must not contain more than 255 characters. Any string class is supported.

# LIB\$RENAME\_FILE

## ***default-filespec***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Default file specification of the files to be renamed. The **default-filespec** argument is the address of a descriptor pointing to the default file specification.

This is an optional argument; if omitted, the default is the null string. See the *VMS Record Management Services Manual* for information on default file specifications. The string must not contain more than 255 characters. Any string class is supported.

## ***related-filespec***

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Related file specification of the files to be renamed. The **related-filespec** argument is the address of a descriptor pointing to the related file specification. This is an optional argument; if omitted, the default is the null string. Any string class is supported.

Input file parsing is used. (See the *VMS Record Management Services Manual* for information on related file specifications and input file parsing.)

The related file specification is useful when you are processing lists of file specifications. Unspecified portions of the file specification are inherited from the last file processed. Any string class is supported. This is an optional argument.

## ***flags***

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Longword of flag bits designating optional behavior. The **flags** argument is the address of an unsigned longword containing the flag bits. This is an optional argument; if omitted, the default is that all flags are clear.

The bit number, symbol, and its meaning is as follows:

---

Bit	Description
0	If <b>new-filespec</b> does not specify a version number, this flag controls whether a new version number for the output file is to be assigned. If clear, the file is given a version number 1 higher than any previously existing file of the same file name and file type. This is the default action. If set, the current version number of the file is used. If a file already exists with the same file name, type and version number, the error RMS\$_FEX is given. This flag is equivalent to the /NONEW_VERSION qualifier of the DCL RENAME command.

---

## ***user-success-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied success routine that LIB\$RENAME\_FILE calls after each successful rename. The **user-success-procedure** argument is the address of the entry mask to the success routine.

For further information on the success routine, see "Call Format for a Success Routine" in the Description section.

## ***user-error-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied error routine that LIB\$RENAME\_FILE calls when it detects an error. The **user-error-procedure** argument is the address of the entry mask to the error routine. The value returned by the error routine determines whether LIB\$RENAME\_FILE processes more files. For further information on the error routine, see "Call Format for an Error Routine" in the Description section.

## ***user-confirm-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied confirm routine that LIB\$RENAME\_FILE calls before it renames a file. The **user-confirm-procedure** argument is the address of the entry mask to the confirm routine. The value returned by the confirm routine determines whether or not LIB\$RENAME\_FILE renames the file.

The confirm routine can be used to select specific files for renaming based on criteria such as expiration date, size, and so on.

For further information on the confirm routine, see "Call Format for a Confirm Routine" in the Description section.

## ***user-specified-argument***

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

Value that LIB\$RENAME\_FILE passes to the success, error, and confirm routines each time they are called. Whatever mechanism is used to pass **user-specified-argument** to LIB\$RENAME\_FILE is also used to pass it to the user-supplied routines. This is an optional argument; if omitted, zero is passed by value.

# LIB\$RENAME\_FILE

## *old-resultant-name*

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

String into which LIB\$RENAME\_FILE copies the old resultant file specification of the last file processed. This is an optional argument. If present, it is used to store the file specification passed to the user-supplied routines instead of a default class S, type T string. Any string class is supported.

If you are specifying one or more of the action routine arguments, be sure that the descriptor class used to pass **resultant-name** is the same as the descriptor class required by the action routine. For example, VAX Ada requires a class SB descriptor for string arguments to Ada routines, but will use a class A descriptor by default when calling external routines. Refer to your language manual to determine the proper descriptor class to use.

## *new-resultant-name*

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

String into which LIB\$RENAME\_FILE writes the new RMS resultant file specification of the last file processed. The **new-resultant-name** argument is the address of a descriptor pointing to the new name. This is an optional argument. If present, it is used to store the file specification passed to the user-supplied routines instead of a class S, type T string. Any string class is supported.

If you are specifying one or more of the action routine arguments, be sure that the descriptor class used to pass **resultant-name** is the same as the descriptor class required by the action routine. For example, VAX Ada requires a class SB descriptor for string arguments to Ada routines, but will use a class A descriptor by default when calling external routines. Refer to your language manual to determine the proper descriptor class to use.

## *file-scan-context*

VMS usage: **context**  
type: **longword (unsigned)**  
access: **modify**  
mechanism: **by reference**

Context for renaming a list of file specifications. The **file-scan-context** is the address of a longword which contains this context. You must initialize this longword to zero before the first of a series of calls to LIB\$RENAME\_FILE. LIB\$RENAME\_FILE uses the file scan context to retain the file context for multiple input files.

LIB\$FILE\_SCAN uses this context to retain multiple input file related file context. This is an optional argument; it need only be specified if you are using multiple input files, as the DCL command RENAME does. You may deallocate the context allocated by LIB\$FILE\_SCAN while processing the LIB\$RENAME\_FILE requests by calling LIB\$FILE\_SCAN\_END after all calls to LIB\$RENAME\_FILE have been completed. See the description of LIB\$FILE\_SCAN for a more detailed description of this argument.

**DESCRIPTION**

This description is divided into three parts.

- Call Format for a Success Routine
- Call Format for an Error Routine
- Call Format for a Confirm Routine

**Call Format for a Success Routine**

The success routine is optional; it is called only if the **user-success-procedure** argument is specified in the call to LIB\$RENAME\_FILE.

The calling format of a success routine is as follows:

```
user-success-procedure  old-filespec ,new-filespec
[,user-specified-argument]
```

**old-filespec**

VMS usage: **char\_string**  
 type: **character string**  
 access: **read only**  
 mechanism: **descriptor**

RMS resultant file specification of the file before it was renamed. If **old-resultant-name** was specified, it is used to pass the string to the success routine. Otherwise, a class S, type T string is passed. Any string class is supported.

**new-filespec**

VMS usage: **char\_string**  
 type: **character string**  
 access: **read only**  
 mechanism: **by descriptor**

RMS resultant file specification of the newly renamed file. If **new-resultant-name** was specified, it is used to pass the string to the success routine. Otherwise, a class S, type T string is passed. Any string class is supported.

**user-specified-argument**

VMS usage: **user\_arg**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **unspecified**

Value of **user-specified-argument** passed by LIB\$RENAME\_FILE to the success routine using the same passing mechanism that was used to pass it to LIB\$RENAME\_FILE.

**Call Format for an Error Routine**

The error routine returns a success/fail value that LIB\$RENAME\_FILE uses to determine whether or not more files will be processed if an error is encountered. The error routine is called only if the **user-error-procedure** argument was specified in the call to LIB\$RENAME\_FILE. If the **user-error-procedure** argument was not specified, the default is to continue processing.

# LIB\$RENAME\_FILE

The calling format of the error routine is as follows:

**user-error-procedure** old-filespec ,new-filespec  
,rms-sts ,rms-stv ,error-source ,user-specified-argument

## **old-filespec**

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

RMS resultant file specification of the file being renamed when the error occurred. If **old-resultant-name** was specified, it is used to pass the string to the error routine. Otherwise, a class S, type T string is passed. Any string class is supported.

## **new-filespec**

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

RMS resultant file specification of the new file name being used when the error occurred. If **new-resultant-name** was specified, it is used to pass the string to the error routine. Otherwise, a class S, type T string is passed. Any string class is supported.

## **rms-sts**

VMS usage: **cond\_value**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Primary condition code (FAB\$L\_STS) which describes the error that occurred. The **rms-sts** argument is the address of an unsigned longword that contains this primary condition code.

## **rms-stv**

VMS usage: **cond\_value**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Secondary condition code (FAB\$L\_STV) which describes the error that occurred. The **rms-stv** argument is the address of an unsigned longword that contains this secondary condition code.

## **error-source**

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Integer code indicating where the error was found. The **error-source** argument is the address of a longword containing the error source.

# LIB\$RENAME\_FILE

The values of **error-source** and their meanings are as follows:

- 0 Error searching for **old-filespec**
- 1 Error parsing **new-filespec**
- 2 Error renaming file

## **user-specified-argument**

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **unspecified**

Value of **user-specified-argument** that LIB\$RENAME\_FILE passes to the error routine using the same passing mechanism that was used to pass it to LIB\$RENAME\_FILE.

If the error routine returns a success status (bit 0 set), then LIB\$RENAME\_FILE will continue processing files. If the error routine returns a failure status (bit 0 clear), processing ceases immediately and LIB\$RENAME\_FILE returns with an error status.

If the **user-error-procedure** argument is not specified, LIB\$RENAME\_FILE will return to its caller the most severe error status encountered while renaming the files. If the error routine is called for an error, the success status LIB\$\_ERRROUCAL is returned.

The error routine is not called for errors related to string copying.

## **Call Format for a Confirm Routine**

The calling format of a confirm routine is as follows:

```
user-confirm-procedure old-filespec ,new-filespec  
,old-fab [,user-specified-argument]
```

### **old-filespec**

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

RMS resultant file specification of the file about to be renamed. If **old-resultant-name** was specified, it is used to pass the string to the confirm routine. Otherwise, a class S, type T string is passed. Any string class is supported.

### **new-filespec**

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

RMS resultant file specification which the file will be given. If **new-resultant-name** was specified, it is used to pass the string to the confirm routine. Otherwise, a class S, type T string is passed. Any string class is supported.



# LIB\$RENAME\_FILE

## old-fab

VMS usage: **fab**  
type: **unspecified**  
access: **read only**  
mechanism: **by reference**

Address of the RMS FAB that describes the file being renamed. Your program may perform an RMS \$OPEN on the FAB to obtain file attributes it needs to determine whether the file should be renamed, but must close the file with \$CLOSE before returning to LIB\$RENAME\_FILE.

## user-specified-argument

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **unspecified**

Value of **user-specified-argument** passed by LIB\$RENAME\_FILE to the confirm routine using the same passing mechanism that was used to pass it to LIB\$RENAME\_FILE. This is an optional argument.

If the confirm routine returns a success value (bit 0 set), the file is renamed; otherwise, the file is not renamed.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_ERRROUCAL	Success—error routine called. A file error was encountered but the error routine was called to handle the condition.
LIB\$_INVARG	Invalid argument. <b>Flags</b> has one or more undefined bits set.
LIB\$_INVFILSPE	Invalid file specification. <b>Old-filespec</b> , <b>new-filespec</b> , or <b>default-filespec</b> contains more than 255 characters.
LIB\$_INVSTRDES	Invalid string descriptor. One of the string argument descriptors was not a valid string descriptor.
LIB\$_WRONUMARG	Wrong number of arguments. An incorrect number of arguments was passed to LIB\$RENAME_FILE.

Any condition value returned by LIB\$SCOPY\_xxx; truncation errors are ignored.

Any condition value returned by RMS. If the **user-error-procedure** argument was not specified, this is the most severe of the RMS errors which occurred while renaming the files.

---

## LIB\$RESERVE\_EF Reserve Event Flag

The Reserve Event Flag routine allocates a local event flag number specified by **event-flag-number**.

---

**FORMAT**            **LIB\$RESERVE\_EF** *event-flag-number*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:        **longword (unsigned)**  
                           access:     **write only**  
                           mechanism: **by value**

---

**ARGUMENT**            ***event-flag-number***  
                           VMS usage: **ef\_number**  
                           type:        **longword (unsigned)**  
                           access:     **read only**  
                           mechanism: **by reference**

Event flag number to be allocated by LIB\$RESERVE\_EF. The **event-flag-number** argument contains the address of a signed longword integer that is this event flag number.

---

**DESCRIPTION**        LIB\$RESERVE\_EF allocates a particular local event flag number. It differs from LIB\$GET\_EF, which allocates an arbitrary event flag.

Use LIB\$FREE\_EF to deallocate an event flag reserved with LIB\$RESERVE\_EF.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_EF_ALRRES	Event flag already reserved.
LIB\$_EF_RESSYS	Event flag reserved to system. This occurs if the event flag number is outside the ranges of 1 to 23 and 32 to 63.

# LIB\$RESERVE\_EF

---

## EXAMPLE

```
PROGRAM RESERVE_EF(INPUT, OUTPUT);  
  
routine LIB$RESERVE_EF(%REF EVENT_FLAG_NUM : INTEGER); EXTERN;  
routine LIB$FREE_EF(%REF EVENT_FLAG_NUM : INTEGER); EXTERN;  
  
VAR  
    FLAG_NUM : INTEGER;  
  
BEGIN  
    FLAG_NUM := 37;  
    LIB$RESERVE_EF(FLAG_NUM);  
    WRITELN(FLAG_NUM);  
    LIB$FREE_EF(FLAG_NUM);  
END.
```

The output generated by this Pascal example program is as follows:

37

---

## LIB\$RESET\_VM\_ZONE Reset Virtual Memory Zone

The Reset Virtual Memory Zone routine frees all blocks of memory that were previously allocated from the zone.

---

**FORMAT**                    **LIB\$RESET\_VM\_ZONE** *zone-id*

---

**RETURNS**                    VMS usage: **cond\_value**  
                                   type:       **longword (unsigned)**  
                                   access:     **write only**  
                                   mechanism: **by value**

---

**ARGUMENTS**                **zone-id**  
                                   VMS usage: **identifier**  
                                   type:       **longword (unsigned)**  
                                   access:     **read only**  
                                   mechanism: **by reference**

Zone identifier. The **zone-id** is the address of a longword that contains the identifier of a zone created by a previous call to LIB\$CREATE\_VM\_ZONE or LIB\$CREATE\_USER\_VM\_ZONE.

---

**DESCRIPTION**                LIB\$RESET\_VM\_ZONE frees all the blocks of memory that were previously allocated from the zone. The memory becomes available to satisfy further allocation requests for the zone; the memory is not returned to the processwide page pool managed by LIB\$GET\_VM\_PAGE. Your program can continue to use the zone after you call LIB\$RESET\_VM\_ZONE.

Resetting a zone is a much more efficient way to reuse storage than individually freeing each allocated object in the zone.

It is the caller's responsibility to ensure that he or she has "exclusive" access to the zone while the reset operation is being performed. Results are unpredictable if another thread of control attempts to perform any operation on the zone while RESET\_VM\_ZONE is in progress.

If you specified deallocation filling when you created the zone, LIB\$RESET\_VM\_ZONE will fill all of the allocated blocks that are freed.

If the zone you are resetting was created using the LIB\$CREATE\_USER\_VM\_ZONE routine, then you must have an appropriate action routine for the reset operation. That is, in your call to LIB\$CREATE\_USER\_VM\_ZONE, you must have specified a **user-reset-procedure**.

# LIB\$RESET\_VM\_ZONE

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Normal successful completion.

LIB\$\_BADBLOADR

An invalid **zone-id** argument.

---

## LIB\$REVERT Revert to the Handler of the Routine Activator

The Revert to the Handler of the Routine Activator routine deletes the condition handler established by LIB\$ESTABLISH by clearing the address pointing to the condition handler from the activated routine's stack frame.

---

### FORMAT LIB\$REVERT

---

#### RETURNS

VMS usage: **address**  
 type: **address**  
 access: **write only**  
 mechanism: **by value**

Previous contents of SF\$A\_HANDLER (longword 0) of the caller's stack frame. This is the address of the condition handler previously in effect. If no condition handler was in effect, zero is returned.

---

#### ARGUMENTS *None.*

---

#### DESCRIPTION

LIB\$REVERT returns the address that it clears from the calling routine's stack frame. LIB\$REVERT is used only if your routine is to establish and then cancel a condition handler for a portion of its execution.

LIB\$REVERT is provided primarily for use with languages without built-in error handling facilities, such as FORTRAN. Do not use LIB\$REVERT from BASIC, COBOL, Pascal, or PL/I. See the documentation for the language you are using for information about how that language handles errors.

In MACRO, you merely use the following instruction rather than calling LIB\$REVERT:

```
CLRL      (FP)      ; set handler address to 0
                        ; in current stack frame
```

---

#### CONDITION VALUES RETURNED *None.*

# LIB\$RUN\_PROGRAM

---

## LIB\$RUN\_PROGRAM Run New Program

The Run New Program routine causes the current program to stop running and begins execution of another program.

---

**FORMAT**            **LIB\$RUN\_PROGRAM** *program-name*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:           **longword (unsigned)**  
                          access:       **write only**  
                          mechanism: **by value**

---

**ARGUMENT**            ***program-name***  
                          VMS usage: **char\_string**  
                          type:           **character string**  
                          access:       **read only**  
                          mechanism: **by descriptor**

File name of the program to be run in place of the current program. The **program-name** argument contains the address of a descriptor pointing to this file name string.

The maximum length of the file name is 255 characters. The default file type is .EXE.

---

**DESCRIPTION**        LIB\$RUN\_PROGRAM stops execution of the current program and begins execution of another program.

- If successful, control does not return to the calling program. Instead, the \$EXIT system service is called, the new program image replaces the old image in the user process, and the command interpreter gives control to the new image.
- If unsuccessful, control returns to the command interpreter.

This routine is supported for use with the DCL and MCR CLIs. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, the error status LIB\$\_NOCLI is returned.

LIB\$RUN\_PROGRAM causes the current image to exit at the point of the call and directs the Command Language Interpreter, if one is present, to start running another program. If LIB\$RUN\_PROGRAM executes successfully, control passes to the second program; if not, control passes to the Command Language Interpreter. The calling program cannot regain control. This technique is called chaining.

This routine is provided primarily for compatibility with PDP-11 systems, where chaining is used to extend the address space of a system.

# LIB\$RUN\_PROGRAM

This routine may also be useful in a VMS environment where address space is severely limited and large images are not possible. For example, you might use chaining to perform system generation (SYSGEN) on a small virtual address space, for a large page file.

With LIB\$RUN\_PROGRAM, the calling program can pass arguments to the next program in the chain only by using the common storage area. One way to do this is for the calling program to call LIB\$PUT\_COMMON to pass the information into the common storage area. Then the called program calls LIB\$GET\_COMMON to retrieve the data.

In general, this practice is not recommended. There is no convenient way to specify the order and type of arguments passed into the common storage area; so programs that pass arguments in this way must know about the format of the data before it is passed. When you use common storage, it is very difficult to keep your program modular and AST-reentrant; a method of arbitration must be designated to define which program can modify common storage and when.

Further, LIB\$RUN\_PROGRAM cannot be used if no Command Language Interpreter is present, as in the case of image subprocesses and detached subprocesses.

If you want control to return to the caller, use LIB\$SPAWN instead.

---

## CONDITION VALUES RETURNED

LIB\$_INVARG	Invalid argument.
LIB\$_NOCLI	No CLI present to perform function. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to DIGITAL in a Software Performance Report (SPR).





---

**DESCRIPTION** LIB\$SCANC uses successive bytes of the string specified by **source-string** to index into a table. The byte selected from the table is the byte on which a logical AND operation is performed with the mask byte. The operation is terminated when the result of the AND operation is equal to 1.

---

**CONDITION  
VALUES  
RETURNED** *None.*

# LIB\$SCOPY\_DXDX

---

## LIB\$SCOPY\_DXDX Copy Source String Passed by Descriptor to Destination

The Copy Source String Passed by Descriptor to Destination routine copies a source string passed by descriptor to a destination string.

---

**FORMAT**                    **LIB\$SCOPY\_DXDX** *source-string ,destination-string*

---

corresponding jsb **LIB\$SCOPY\_DXDX6**  
entry point

---

**RETURNS**                    VMS usage: **cond\_value**  
                                  type:            **longword (unsigned)**  
                                  access:        **write only**  
                                  mechanism:    **by value**

---

**ARGUMENTS**                ***source-string***  
                                  VMS usage: **char\_string**  
                                  type:        **character string**  
                                  access:      **read only**  
                                  mechanism: **by descriptor**

Source string to be copied to the destination string by LIB\$SCOPY\_DXDX. The **source-string** argument contains the address of a descriptor pointing to this source string. The descriptor class can be unspecified, fixed-length, decimal string, array, noncontiguous array, varying, or dynamic.

***destination-string***  
VMS usage: **char\_string**  
type:        **character string**  
access:      **write only**  
mechanism: **by descriptor**

Destination string to which the source string is copied. The **destination-string** argument contains the address of a descriptor pointing to this destination string.

The following actions occur depending on the class of the destination string's descriptor.

# LIB\$SCOPY\_DXDX

<b>Class Field</b>	<b>Action</b>
DSC\$K_CLASS_S,Z,SD,A,NCA	Copy the source string. If needed, space-fill or truncate on the right.
DSC\$K_CLASS_D	If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor. If the area specified is not large enough, return the previous space allocation (if any) and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor.
DSC\$K_CLASS_VS	Copy source string to destination string up to the limit of DSC\$W_MAXSTRLEN with no padding. Readjust current length field to actual number of bytes copied.

## DESCRIPTION

LIB\$SCOPY\_DXDX returns all condition values as a status; truncation is a qualified success condition value (bit 0 set to 1).

In addition, an equivalent JSB entry point is available, with R0 containing the first argument and R1 containing the second.

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. All characters in the input string were copied to the destination string.
LIB\$_STRTRU	Routine successfully completed. String truncated. The fixed-length destination string could not contain all of the characters copied from the source string.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$B_CLASS field.



# LIB\$SCOPY\_R\_DX

The following actions occur depending on the class of the destination string's descriptor.

Class Field	Action
DSC\$K_CLASS_S,Z,SD,A,NCA	Copy the source string. If needed, space fill or truncate on the right.
DSC\$K_CLASS_D	If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor. If the area specified is not large enough, return the previous space allocation (if any) and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor.
DSC\$K_CLASS_VS	Copy source string to destination string up to the limit of DSC\$W_MAXSTLEN with no padding. Readjust current length field to actual number of bytes copied.

## DESCRIPTION

LIB\$SCOPY\_R\_DX returns all condition values as a status; truncation is a qualified success condition value (bit 0 set to 1).

In addition, an equivalent JSB entry is available, with R0 being the first argument, R1 the second, and R2 the third. The length argument is passed in bits 15:0 of R0.

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. All characters in the input string were copied to the destination string.
LIB\$_STRTRU	Routine successfully completed. String truncated. The fixed-length destination string could not contain all of the characters copied from the source string.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$B_CLASS field.



**attributes**

VMS usage: **mask\_longword**  
 type: **longword (unsigned)**  
 access: **read only**  
 mechanism: **by reference**

Logical name or translation attributes. The **attributes** argument is the address of a longword bit mask that contains the logical name or translation attributes.

LNMSM\_CONFINE and LNMSM\_NO\_ALIAS are currently available logical name attributes. See the description of the \$CRELNM system service in the *VMS System Services Reference Manual* for definitions of LNMSM\_CONFINE and LNMSM\_NO\_ALIAS. If omitted, no special logical name attribute is established.

If no **item-list** is specified, the translation attributes LNMSM\_CONCEALED and LNMSM\_TERMINAL may be specified. See the description of the ASSIGN command in the *VMS DCL Dictionary* for definitions of these attributes. If an **item-list** is specified, it will contain the translation attributes for each equivalence string in the attribute.

**item-list**

VMS usage: **item\_list\_2**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **by reference, array reference**

Item list describing the equivalence names for this logical name. The **item-list** argument contains the address of an array that contains this item list. If **item-list** is not specified, the logical name will have only one value, as specified in the **value-string** argument.

Either **value-string** or **item-list** must be specified. If neither is specified, the LIB\$\_INVARG error is produced. If both **value-string** and **item-list** are specified, the **value-string** argument is ignored.

If **item-list** is specified, only logical name attributes are permitted. Translation attributes appear in the item list.

**Item-list** is only needed when you wish to create multiple equivalence strings for a single logical name.

---

**DESCRIPTION**

If the optional **table** argument is defined, the logical name will be placed in the table specified by the **table** argument; otherwise, the logical name is placed in the LNM\$PROCESS table.

Unlike the system services \$CRELOG and \$CRELNM, LIB\$SET\_LOGICAL does not require the caller to be executing in supervisor mode to define a supervisor-mode logical name. Supervisor-mode logical names are not deleted when an image exits. A program can obtain the current value of any logical name by calling the system service \$TRNLNM. For more information on logical names see the *VMS System Services Reference Manual*.

This routine is supported for use with the DCL and MCR Command Language Interpreters. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In that case, the error status LIB\$\_NOCLI is returned.

See the *VMS DCL Dictionary* for a description of the DCL DEFINE command.



# LIB\$SET\_LOGICAL

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
SS\$_SUPERSEDE	Routine successfully completed; the previous definition of the logical name was replaced.
SS\$_BUFFEROVF	Routine successfully completed; however, a buffer overflow occurred.
SS\$_ACCVIO	Access violation. The logical name or its value could not be read.
SS\$_BADPARAM	Bad argument.
SS\$_INSFMEM	Insufficient dynamic memory.
SS\$_IVLOGNAM	Invalid logical name. The logical name or its value contained more than 255 characters.
SS\$_IVLOGTAB	Invalid logical name table.
SS\$_NOPRIV	No privileges for attempted operation.
SS\$_TOOMANYLNAM	Logical name translation exceeded allowed depth.
LIB\$_INVARG	Neither the <b>value-string</b> nor the <b>item-list</b> argument was specified.
LIB\$_INVSTRDES	Invalid string descriptor.
LIB\$_NOCLI	No CLI present to perform function. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL Command Language Interpreter, please report the problem to DIGITAL in a Software Performance Report (SPR).

---

## EXAMPLE

```
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
1234567890123456789012345678901234567890123456789012345678901234567890
```

```
C* Initialize name for logical
C          MOVE 'RPG_LOG' LOGICL 7
C* Initialize value to which logical is to be set
C          MOVE 'OFF' SETVAL 3
C          SETLOG EXTRN'LIB$SET_LOGICAL'
C* Call RTL routine to set the logical
C          CALL SETLOG
C          PARM      LOGICL
C          PARM      SETVAL
C          SETON          LR
```

The RPG II program above sets the logical RPG\_LOG to OFF. This value can be displayed after the program is run with SHOW LOGICAL as follows:

```
$ SHOW LOGICAL RPG_LOG
"RPG_LOG" = "OFF" (LNM$PROCESS_TABLE)
```



# LIB\$SET\_SYMBOL

## ***table-type-indicator***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by reference**

Indicator of the table which will contain the defined symbol. The **table-type-indicator** argument is the address of a signed longword integer that is this table indicator.

If omitted, the local symbol table is used. The following are possible values for **table-type-indicator**.

Symbolic Name	Value	Table Used
LIB\$K_CLI_LOCAL_SYM	1	Local symbol table
LIB\$K_CLI_GLOBAL_SYM	2	Global symbol table

## **DESCRIPTION**

LIB\$SET\_SYMBOL requests the calling process's Command Language Interpreter (CLI) to define or redefine a CLI symbol.

CLI symbols created using LIB\$SET\_SYMBOL may be inaccessible by other CLI commands. To avoid this situation, make sure that your symbol names are alphanumeric and that the first character is alphabetic. LIB\$SET\_SYMBOL is intended to set string CLI symbols, not integer CLI symbols.

LIB\$K\_CLI\_LOCAL\_SYM and LIB\$K\_CLI\_GLOBAL\_SYM are defined as global symbols and in DIGITAL-supplied symbol libraries (macro or module name \$LIBCLIDEF).

This routine is supported for use with the DCL Command Language Interpreter. If used with the MCR CLI, the error status LIB\$\_NOCLI will be returned. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In this case, the error status LIB\$\_NOCLI is returned.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_AMBSYMDDEF	Ambiguous symbol definition. The symbol name you want to define is ambiguous when compared with existing symbol names. This condition might arise if abbreviated symbols have been defined previously. See the <i>VMS DCL Dictionary</i> for more information on abbreviated symbols.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSCLIMEM	Insufficient CLI memory. The CLI could not get enough virtual memory to assign another symbol. This condition may be caused by having too many symbols defined; deleting some symbol definitions may make enough room for the new symbol.
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVARG	Invalid argument. The value of <b>table-type-indicator</b> was invalid or the length of <b>value-string</b> was greater than 255 characters.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.
LIB\$_INVSYMNAM	Invalid symbol name. The length of <b>symbol</b> was greater than 255 characters or <b>symbol</b> did not begin with a letter.
LIB\$_NOCLI	No CLI present to perform function. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
LIB\$_UNECLIERR	Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL Command Language Interpreter, please report the problem to DIGITAL in a Software Performance Report (SPR).

# LIB\$SET\_SYMBOL

---

## EXAMPLE

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  
1234567890123456789012345678901234567890123456789012345678901234567890

```
C* Initialize name for symbol
C          MOVE 'RPG_SYM' SYM  7
C* Initialize value to which symbol is to be set
C          MOVE 'ON'          SETVAL 2
C          SETSYM  EXTRN'LIB$SET_SYMBOL'
C* Call RTL routine to set the symbol
C          CALL SETSYM
C          PARM  SYMBOL
C          PARM  SETVAL
C          SETON                               LR
```

The RPG II program above sets the symbol RPG\_SYM to ON. This value can be displayed after the program is run with SHOW SYMBOL as follows:

```
$ SHOW SYMBOL RPG_SYM
RPG_SYM = "ON"
```

---

## LIB\$SFREE1\_DD Free One Dynamic String

The Free One Dynamic String routine returns one dynamic string area to free storage.

---

**FORMAT**                    **LIB\$SFREE1\_DD** *descriptor-address*

---

corresponding jsb entry point    **LIB\$SFREE1\_DD6**

---

**RETURNS**                    VMS usage: **cond\_value**  
                                   type:            **longword (unsigned)**  
                                   access:        **write only**  
                                   mechanism:    **by value**

---

**ARGUMENT**                    ***descriptor-address***  
                                   VMS usage: **quadword\_unsigned**  
                                   type:        **quadword (unsigned)**  
                                   access:      **modify**  
                                   mechanism: **by reference**

Dynamic descriptor specifying the area to be deallocated. The **descriptor-address** argument is the address of an unsigned quadword that is this descriptor. The descriptor is assumed to be dynamic and its class field is not checked.

---

**DESCRIPTION**                Before a routine deallocates a dynamic descriptor, it must use LIB\$SFREE1\_DD or LIB\$SFREE<sub>n</sub>\_DD to deallocate the string storage space specified by the dynamic descriptor. Otherwise, string storage is not deallocated and your program can run out of memory.

This routine deallocates the described string space and flags the descriptor as describing no string at all (DSC\$A\_POINTER = 0 and DSC\$W\_LENGTH = 0).

---

<b>CONDITION VALUES RETURNED</b>	SS\$_NORMAL	Routine successfully completed.
	LIB\$_FATERRLIB	Fatal internal error.



---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL  
LIB\$\_FATERRLIB

Routine successfully completed.  
Fatal internal error.



# LIB\$\$GET1\_DD

---

## LIB\$\$GET1\_DD Get One Dynamic String

The Get One Dynamic String routine allocates dynamic virtual memory to the string descriptor you specify.

---

**FORMAT**                    **LIB\$\$GET1\_DD** *word-integer-length , descriptor-part*

---

corresponding jsb   **LIB\$\$GET1\_DD\_R6**  
entry point

---

**RETURNS**                VMS usage: **cond\_value**  
                              type:            **longword (unsigned)**  
                              access:        **write only**  
                              mechanism:    **by value**

---

**ARGUMENTS**            ***word-integer-length***  
                              VMS usage: **word\_unsigned**  
                              type:            **word (unsigned)**  
                              access:        **read only**  
                              mechanism:    **by reference**

Number of bytes of dynamic virtual memory to be allocated by LIB\$\$GET1\_DD. The **word-integer-length** argument is the address of an unsigned word that contains this number. The amount of storage allocated may be rounded up automatically. If the number of bytes is zero, a small amount of space is allocated.

***descriptor-part***  
VMS usage: **quadword\_unsigned**  
type:            **quadword (unsigned)**  
access:        **write only**  
mechanism:    **by reference**

Descriptor of the dynamic string to which LIB\$\$GET1\_DD will allocate the dynamic virtual memory. The **descriptor-part** argument contains the address of an unsigned quadword that is this descriptor.

The **descriptor-part** argument must contain the address of a dynamic string descriptor; LIB\$\$GET1\_DD returns an unpredictable result if any other type of descriptor is specified by this argument.

The class field is not checked but is set to dynamic (DSC\$B\_CLASS = 2). The length field (DSC\$W\_LENGTH) is set to **word-integer-length**, and the address field (DSC\$A\_POINTER) points to the string area allocated.

---

**DESCRIPTION**

LIB\$SGET1\_DD is similar to LIB\$SCOPY\_DXDX except that no source string is copied. You can write anything you want in the allocated area.

If **descriptor-part** already has dynamic memory allocated to it, but the amount allocated is less than **word-integer-length**, that space is deallocated before LIB\$SGET1\_DD allocates new space.

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Routine successfully completed.

LIB\$\_FATERRLIB

Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).

LIB\$\_INSVIRMEM

Insufficient virtual memory. A call to LIB\$GET\_VM has failed because your program has exceeded the image quota for virtual memory.



# LIB\$SHOW\_TIMER

The following values are allowed for the **code** argument.

Value	Description
1	Elapsed time
2	CPU time
3	Buffered I/O
4	Direct I/O
5	Page faults

## ***user-action-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

User-supplied action routine called by LIB\$SHOW\_TIMER. The **user-action-procedure** argument is the address of the entry mask to this routine. The default action of LIB\$SHOW\_TIMER is to write the results to SYS\$OUTPUT. An action routine is useful if you want to write the results to a file, or in general, anywhere other than SYS\$OUTPUT.

The action routine returns either a success or failure condition value; this status is returned to the calling program as the value of LIB\$SHOW\_TIMER.

## ***user-argument-value***

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

A 32-bit value to be passed to the action routine without interpretation. If omitted, LIB\$SHOW\_TIMER passes a zero by value to the user routine.

---

## **DESCRIPTION**

LIB\$SHOW\_TIMER returns the times and counts accumulated since the last call to LIB\$INIT\_TIMER. By default, when neither **code** nor **user-action-procedure** is specified in the call, LIB\$SHOW\_TIMER writes to SYS\$OUTPUT a line giving the information listed below.

Shown on Line	Description
ELAPSED = dddd hh:mm:ss.cc	Elapsed real time
CPU = hhhh:mm:ss.cc	Elapsed CPU time
BUFIO = nnnn	Count of buffered I/O operations
DIRIO = nnnn	Count of direct I/O operations
PAGEFAULTS = nnnn	Count of page faults

Any one or all five statistics can be written to SYS\$OUTPUT or passed to your user-supplied action routine for other processing.

# LIB\$SHOW\_TIMER

## Call Format for an Action Routine

Action routine is a user-supplied routine called by LIB\$SHOW\_TIMER. The action routine is used when you wish to write results to anywhere other than SYS\$OUTPUT. The action routine is called only when you specify the **user-action-procedure** argument in the call to LIB\$SHOW\_TIMER.

LIB\$SHOW\_TIMER calls the action routine using this format:

```
user-action-procedure out-str [,user-argument-value]
```

### out-str

VMS usage: **char\_string**  
type: **character string**  
access: **read only**  
mechanism: **by descriptor**

Fixed-length string containing the statistics requested. The string is formatted exactly as it would be if written to SYS\$OUTPUT. The leading character is blank.

### user-argument-value

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

A 32-bit value passed to LIB\$SHOW\_TIMER. The user argument is passed without interpretation to the action routine.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_INVARG	Invalid argument. Either <b>code</b> or <b>handle-address</b> was invalid.

Any condition values returned by LIB\$PUT\_OUTPUT or your action routine.

---

## EXAMPLE

```
PROGRAM SHOW_TIMER(INPUT,OUTPUT);  
{+}  
{ This Pascal example demonstrates how to use LIB$SHOW_TIMER.  
{-}  
  
  VAR  
    RETURNED_STATUS : INTEGER;  
  
  [EXTERNAL] FUNCTION LIB$INIT_TIMER(  
    HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0  
  ) : INTEGER; EXTERNAL;
```

# LIB\$SHOW\_TIMER

```
[EXTERNAL] FUNCTION LIB$SHOW_TIMER(  
    HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0;  
    CODE       : INTEGER;  
    [IMMEDIATE,UNBOUND]  
    ROUTINE ACTION_RTN( OUT_STR : [CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;  
                        USER_ARG : INTEGER) := %IMMED 0;  
    USER_ARG   : INTEGER := %IMMED 0  
    ) : INTEGER; EXTERNAL;  
  
[EXTERNAL] FUNCTION LIB$STOP(  
    CONDITION_STATUS : [IMMEDIATE,UNSAFE] UNSIGNED;  
    FAO_ARGS         : [IMMEDIATE,UNSAFE,LIST] UNSIGNED  
    ) : INTEGER; EXTERNAL;  
  
ROUTINE USER_ACTION_RTN(  
    OUT_STR : [CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;  
    USER_ARG : INTEGER);  
  
    BEGIN  
        WRITELN('User argument is ',USER_ARG:1);  
        WRITELN(OUT_STR);  
    END;  
  
BEGIN  
  
{+}  
{ Call LIB$INIT_TIMER to initialize RTL internal counters.  
{-}  
  
RETURNED_STATUS := LIB$INIT_TIMER;  
IF NOT ODD(RETURNED_STATUS)  
THEN  
    LIB$STOP(RETURNED_STATUS);  
  
{+}  
{ Print a line of text to waste time.  
{-}  
  
WRITELN('Spend time to acquire elapsed real time and page faults');  
  
{+}  
{ Call LIB$SHOW_TIMER to display counters.  
{-}  
  
RETURNED_STATUS := LIB$SHOW_TIMER(,0,USER_ACTION_RTN,5);  
END.
```

This Pascal program demonstrates how to call LIB\$SHOW\_TIMER. The output generated by this Pascal example is as follows:

```
$ RUN SHOW_TIMER  
Spend time to acquire elapsed real time and page faults  
User argument is 5  
ELAPSED: 0 00:00:00.44 CPU: 0:00:00.04  
BUFIO: 1 DIRIO: 0 FAULTS: 18
```



***user-action-procedure***

VMS usage: **procedure**  
 type: **procedure entry mask**  
 access: **function call (before return)**  
 mechanism: **by value**

User-supplied action routine called by LIB\$SHOW\_VM. By default, LIB\$SHOW\_VM returns statistics to SYS\$OUTPUT. An action routine is useful when you want to return statistics to a file or, in general, to any place other than SYS\$OUTPUT. The **user-action-procedure** argument is the address of the entry mask to the action routine. The routine returns either a success or failure condition value, which will be returned as the value of LIB\$SHOW\_VM.

For more information on the action routine, see "Call Format for an Action Routine" in the Description section.

***user-specified-argument***

VMS usage: **user\_arg**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **by value**

A 32-bit value to be passed directly to the action routine without interpretation. That is, the contents of the argument list entry **user-specified-argument** are copied to the argument list entry for **user-action-procedure**.

---

**DESCRIPTION**

LIB\$SHOW\_VM returns the statistics accumulated from calls to LIB\$GET\_VM/LIB\$FREE\_VM and LIB\$GET\_VM\_PAGE/LIB\$FREE\_VM\_PAGE. By default, with neither **code** nor **user-action-procedure** specified in the call, LIB\$SHOW\_VM writes a line giving the following information to SYS\$OUTPUT:

mmm calls to LIB\$GET\_VM, nnn calls to LIB\$FREE\_VM, ppp bytes still allocated

Optionally, any one of six statistics can be output to SYS\$OUTPUT and/or the line of information can be passed to a user-specified "action routine" for processing different from the default.

**Call Format for an Action Routine**

The action routine is a user-supplied routine that LIB\$SHOW\_VM calls if you specify the **user-action-procedure** argument in the call to LIB\$SHOW\_VM.

The call format for an action routine is:

**user-action-procedure** resultant-string ,user-specified-argument

**resultant-string**

VMS usage: **char\_string**  
 type: **character string**  
 access: **write only**  
 mechanism: **by descriptor**

Statistics supplied by LIB\$SHOW\_VM. The **resultant-string** argument is the address of a descriptor pointing to an address into which LIB\$SHOW\_VM writes the statistics. The string is formatted exactly as it would be if written to SYS\$OUTPUT. The first character is a blank; carriage-return/line-feed combinations are not included.



# LIB\$SHOW\_VM

## **user-specified-argument**

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

The 32-bit value passed to LIB\$SHOW\_VM is passed to the action routine without interpretation. If the **user-specified-argument** argument is omitted in the call to LIB\$SHOW\_VM, a zero is passed by value to the user routine.

---

## **CONDITION VALUES RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_INVARG	Invalid arguments. This can be caused by an invalid value for <b>code</b> .

Any condition values returned by LIB\$PUT\_OUTPUT or your action routine.



# LIB\$SHOW\_VM\_ZONE

## ***user-action-procedure***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **function call (before return)**  
mechanism: **by value**

Optional user-supplied action routine called by LIB\$SHOW\_VM\_ZONE. The **user-action-procedure** argument is the address of the entry mask of the action routine. By default, LIB\$SHOW\_VM\_ZONE prints statistics to SYS\$OUTPUT via LIB\$PUT\_OUTPUT. An action routine is useful when you want to return statistics to a file or, in general, to any location other than SYS\$OUTPUT. If **user-action-procedure** fails, LIB\$SHOW\_VM\_ZONE terminates and returns a failure code. Success codes are ignored.

For more information on the action routine, see the Description section.

## ***user-arg***

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

Optional 32-bit value to be passed directly to the action routine without interpretation. That is, the contents of the argument list entry **user-arg** are copied to the argument list entry for **user-action-procedure**.

---

## **DESCRIPTION**

LIB\$SHOW\_VM\_ZONE returns formatted information about the specified zone and passes it to the action routine. The **detail-level** argument determines the degree of detail of the zone information returned, and this information is formatted into a readable display and passed to either a user action routine or to LIB\$PUT\_OUTPUT.

The action routine is a user-supplied routine that LIB\$SHOW\_VM\_ZONE calls if you specify the **action-routine** argument in the call to LIB\$SHOW\_VM\_ZONE. If you do not specify **action-routine**, the information is passed to LIB\$PUT\_OUTPUT for output to SYS\$OUTPUT. The call format for an action routine is as follows:

```
action-routine string, user-arg
```

### **Arguments**

#### **string**

VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

Information supplied by LIB\$SHOW\_VM\_ZONE. The **string** argument is the address of a descriptor pointing to an address into which LIB\$SHOW\_VM\_ZONE writes the requested information. The string is formatted exactly as it would be if written to SYS\$OUTPUT.

# LIB\$SHOW\_VM\_ZONE

## user-arg

VMS usage: **user\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

The 32-bit value passed to LIB\$SHOW\_VM\_ZONE is passed to the action routine without interpretation. If the **user-arg** argument is omitted in the call to LIB\$SHOW\_VM\_ZONE, a zero is passed by value to the user routine.

If no **zone-id** is specified (0 is passed), the default zone is used.

You must ensure that you have exclusive access to the zone while information is being displayed. Results are unpredictable and may be inconsistent if another thread of control modifies the zone while this routine is displaying data or scanning control blocks.

While scanning the queues and free lists, this routine may detect errors.

If the lookaside list summary discovers a block improperly linked into the list so that the list appears disjointed, the count of the number of blocks of that particular size will be displayed as asterisks.

In addition, the following errors and warnings may be displayed during the lookaside list and area free list scans. The format is as follows:

```
**** ERROR -- error description ****  
**** WARNING -- warning description ****
```

Error Message	Description
Invalid block size	The size of the block is either not large enough to contain the necessary queue links or is unreasonably large. The size field has been corrupted, therefore the size of the block is reduced so the block to be dumped fits within the area.
Block not owned by zone	The current block is not within a section of the virtual address space controlled by this zone. It is possibly attempting to free a block not originally allocated from this zone.
Block extends past the end of area; truncated	The end of the block is not in the area from which the block has been allocated. The size field may have been corrupted, therefore the size of the block is reduced so the block to be dumped fits within the area.
Block extends into "unallocated" block, truncated	The end of the block extends past the allocated section of the area. The size field may have been corrupted, therefore the size of the block is reduced so the block to be dumped fits within the area.

# LIB\$SHOW\_VM\_ZONE

Error Message	Description
Current block not completely accessible	The current block extends into a nonexistent part of the virtual address space. The size field may have been corrupted, therefore the size of the block is reduced so the block to be dumped fits within the area.
Back link does not return to previous block	The back link in a doubly linked list does not point to the previous block.
Forward link does not point to valid address	The forward link of current block points to a location that is not in the virtual address space.
Free-fill mismatch	One of the locations filled when the block was freed has been modified.
Boundary tag mismatch	One of the boundary tags of the block is not valid.

Warning	Description
Forward link of current block may not be valid	The back link of the block pointed to by the forward link of the current block does not point to the current block.
Block at nnnnnnnn is not accessible	The block at location nnnnnnnn could not be accessed and cannot be dumped.
Block truncated to nnnnnnnn bytes to prevent ACCVIO	The block to be dumped extends into the unaccessible part of the address space. The size of the block is reduced such that the block to be dumped fits within the accessible addresses.

When a block forward link is suspected of pointing to an invalid next block, the information from the next block is replaced by asterisks. The following is a sample error display.

\*\*\*\* ERROR -- forward link does not point to valid address \*\*\*\*

Link Analysis for Current Block:

	Previous	Current	Next
	-----	-----	-----
Block adr :	0014B270	0014C200	6B6E754A
Forw link (abs):	0014C200	6B6E754A	*****

Block size = 32

Block contents:

```
00000000 00000000 6B6E754A 00000020 ...Junk..... 00000 0014C200
0014B270 00000008 00000000 00000000 .....p.. 00010 0014C210
```

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Normal successful completion.
LIB\$_NOTFOU	Could not find another VM zone (alternate success status).
LIB\$_BADZONE	Invalid zone. Routine was called with a <b>zone-id</b> that does not represent a valid VM zone.
LIB\$_INVARG	Invalid argument.
LIB\$_INVOPEZON	Invalid operation for zone; invalid use of unspecified user zone action routine.
LIB\$_WRONUMARG	Wrong number of arguments.

Any condition value returned by the user-formatted output action routine or LIB\$PUT\_OUTPUT.

---

**EXAMPLE**

```

IMPLICIT NONE
INTEGER*4 zone_id
INTEGER*4 LIB$SHOW_VM_ZONE

zone_id = 0           ! request info for default zone
call LIB$SHOW_VM_ZONE (zone_id, 1)
END

```

An example of the output generated by this FORTRAN program using **detail-level 1** is as follows:

```

ZONE_ID = 00013058,  ZONE_NAME = "DEFAULT_ZONE"

Algorithm = LIB$K_VM_FIRST_FIT
Flags = 00000020
      LIB$M_VM_EXTEND_AREA

Initial size = 124 pages      Current size = 0 pages
Extend size = 128 pages      Page limit = 0 pages

Requests are rounded up to a multiple of 8 bytes,
naturally aligned on 8-byte boundaries

0 bytes freed and not yet reallocated

```

```

IMPLICIT NONE
INTEGER*4 zone_id
INTEGER*4 LIB$SHOW_VM_ZONE

zone_id = 0           ! request info for default zone
call LIB$SHOW_VM_ZONE (zone_id, 3)
END

```

# LIB\$SHOW\_VM\_ZONE

An example of the output generated by this FORTRAN program using detail-level 3 is as follows:

Zone Id = 00044CF8, Zone name = "Mix of lookaside list and area blocks"

Algorithm = LIB\$K\_VM\_QUICK\_FIT with 16 Lookaside Lists ranging from a minimum blocksize of 8, to a maximum blocksize of 128

Flags = 00000028

LIB\$M\_VM\_FREE\_FILLO  
LIB\$M\_VM\_EXTEND\_AREA

Initial size = 16 pages Current size = 256 pages in 1 area  
Extend size = 16 pages Page limit = None

Requests are rounded up to a multiple of 8 bytes,  
naturally aligned on 8-byte boundaries

129896 bytes have been freed and not yet reallocated

312 bytes are used for zone and area control blocks, or 0.2% overhead

Quick Fit Lookaside List Summary:

List number	Block size	Number of blocks
2	16	7
3	24	9
4	32	8
5	40	6
6	48	12
7	56	6
8	64	7
9	72	7
10	80	5
11	88	4
12	96	8
13	104	3
14	112	5
15	120	12
16	128	6

Area Summary:

First address	Last address	Pages assigned	Bytes not yet allocated
00065400	000853FF	256	1176

Scanning Lookaside Lists in Zone Control Block

Scanning Free List for Area at 00065400

Number of blocks = 84, Min blocksize = 160, Max blocksize = 5768





# LIB\$SIGNAL

## ***FAO-argument1***

VMS usage: **varying\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

Optional FAO (formatted ASCII output) argument that is associated with the specified condition value.

Section 4.1.5 explains the message format.

## ***condition-value2***

VMS usage: **cond\_value**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by value**

VAX 32-bit condition value. The optional **condition-value2** argument is an unsigned longword that contains this condition value.

Section 4.1.2 explains the format of a VAX condition value.

## ***number-of-arguments2***

VMS usage: **longword\_signed**  
type: **longword integer (signed)**  
access: **read only**  
mechanism: **by value**

Number of FAO arguments associated with the condition value. The optional **number-of-arguments2** argument is a signed longword integer that contains this number. If omitted or specified as zero, no FAO arguments follow.

## ***FAO-argument2***

VMS usage: **varying\_arg**  
type: **unspecified**  
access: **read only**  
mechanism: **by value**

Optional FAO (formatted ASCII output) argument that is associated with the specified condition value.

Section 4.1.5 explains the message format.

---

## **DESCRIPTION**

Your program calls LIB\$SIGNAL whenever it needs to indicate an exception condition or output a message rather than return a status code to its calling program.

LIB\$SIGNAL examines the primary and secondary exception vectors and then scans the stack, frame by frame, starting at the top of the stack, and calls each condition handler it finds. LIB\$SIGNAL locates stack frames by using each frame's saved frame pointer (FP) to chain back through the stack frames. Section 4.1.3 provides additional information on this process.

If one of the handlers that LIB\$SIGNAL calls returns a continue code (that is, any success completion code with bit 0 set to 1), LIB\$SIGNAL returns to its caller, which should be prepared to continue execution.

If the handler that LIB\$SIGNAL calls returns a resignal code (that is, any completion code with bit 0 set to 0) LIB\$SIGNAL continues to scan the stack.

If the handler called by LIB\$SIGNAL calls SYS\$UNWIND, control will not return to LIB\$SIGNAL's caller, thus changing the program flow. A handler can also modify the saved copy of R0/R1 in the mechanism vector, changing registers R0 and R1 after the stack has been unwound. If a handler does neither of these things, then all registers including R0/R1 and the hardware condition codes are preserved.

LIB\$SIGNAL will, if necessary, scan up to 65,536 previous stack frames and then finally examine the last-chance exception vector.

The LIB\$SIGNAL argument list, the Program Counter (PC) and Processor Status Longword (PSL) of the caller are appended to build the signal argument vector.

---

**CONDITION VALUES RETURNED**      *None.*

---

## EXAMPLES



```
C+
C This FORTRAN example program demonstrates the use of
C LIB$SIGNAL.
C
C This program defines SS$... signals and then calls LIB$SIGNAL
C passing the access violation code as the argument.
C-
```

```
INCLUDE '($SDEF)'
CALL LIB$SIGNAL ( %VAL(SS$_ACCVIO) )
END
```

In FORTRAN, this code fragment signals the standard system message ACCESS VIOLATION.

The output generated by this FORTRAN program is as follows:

```
%SYSTEM-F-ACCVIO, access violation, reason mask=10, virtual address=03C00020,
PC=00000000, PSL=08000000
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name      routine name      line      rel PC      abs PC
D2$MAIN          D2$MAIN          683       00000010   00000410
```

# LIB\$SIGNAL

2

```
;+
; This MACRO example program demonstrates the use of LIB$SIGNAL
; by forcing an access violation to be signaled.
;-
.EXTRN SS$_ACCVIO ; Declare external symbol
.ENTRY START,0
PUSHL #SS$_ACCVIO ; Condition value symbol
; for access violation
CALLS #1, G^LIB$SIGNAL ; Signal the condition
RET
.END START

.EXTRN SS$_ACCVIO ; Declare external symbol
PUSHL #SS$_ACCVIO ; Condition value symbol
; for access violation
CALLS #1, LIB$SIGNAL ; Signal the condition
```

This example shows the equivalent MACRO code. The output generated by this program is as follows:

```
%SYSTEM-F-ACCVIO, access violation, reason mask=0F, virtual address=03C00000,
PC=00000000, PSL=00000000
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name      routine name      line      rel PC      abs PC
.MAIN.           START                0000000F  0000020F
```



# LIB\$SIG\_TO\_RET

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Routine successfully completed; SS\$\_UNWIND  
completed. Otherwise, the error code from  
SS\$\_UNWIND is returned.

---

## EXAMPLE

```
C+
C This FORTRAN example demonstrates how to use LIB$SIG_TO_RET.
C
C This function subroutine inverts each entry in an array. That is,
C a(i,j) becomes 1/a(i,j). The subroutine has been declared as an integer
C function so that the status of the inversion may be returned. The status
C should be success, unless one of the a(i,j) entries is zero. If one of
C the a(i,j) = 0, then 1/a(i,j) is division by zero. This division by zero
C does not cause a division by zero error, rather, the routine will return
C signal a failure.
C-
```

```
      INTEGER*4 FUNCTION FLIP(A,N)
      DIMENSION A(N,N)
      EXTERNAL LIB$SIG_TO_RET
      CALL LIB$ESTABLISH (LIB$SIG_TO_RET)
      FLIP = .TRUE.
```

```
C+
C Flip each entry.
C-
```

```
      DO 1 I = 1, N
      DO 1 J = 1, N
1      A(I,J) = 1.0/A(I,J)
      RETURN
      END
```

```
C+
C This is the main code.
C-
```

```
      INTEGER STATUS, FLIP
      REAL ARRAY_1(2,2),ARRAY_2(3,3)
      DATA ARRAY_1/1,2,3,4/,ARRAY_2/1,2,3,5,0,5,6,7,2/
      CHARACTER*32 TEXT(2),STRING
      DATA TEXT(1)/' This array could be flipped.  '/,
1      TEXT(2)/' This array could not be flipped.'/

      STRING = TEXT(1)
      STATUS = FLIP(ARRAY_1,2)
      IF ( .NOT. STATUS) STRING = TEXT(2)
      TYPE '(a)',      STRING

      STRING = TEXT(1)
      STATUS = FLIP(ARRAY_2,3)
      IF ( .NOT. STATUS) STRING = TEXT(2)
      TYPE '(a)',      STRING

      END
```

This FORTRAN example program inverts each entry in an array. The output generated by this program is as follows:

This array could be flipped.  
This array could not be flipped.



---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL

Routine successfully completed; SS\$\_UNWIND completed. Otherwise, the error code from SS\$\_UNWIND is returned.

LIB\$\_INVARG

Invalid argument. The condition code in **signal-arguments** is SS\$\_UNWIND.





See the *VAX Architecture Reference Manual* for more information on faults and traps.

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_RESIGNAL

Resignal condition to next handler. The exception was not one that LIB\$SIM\_TRAP could handle.

# LIB\$SKPC

---

## LIB\$SKPC Skip Equal Characters

The Skip Equal Characters routine compares each character of a given string with a given character and returns the relative position of the first nonequal character as an index. LIB\$SKPC makes the VAX SKPC instruction available as a callable routine.

---

**FORMAT**            **LIB\$SKPC** *character-string ,source-string*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:        **write only**  
                          mechanism:    **by value**

The relative position in the source string of the first unequal character. LIB\$SKPC returns a zero if the source string was of zero length or if every character in **source-string** was equal to **character-string**.

---

**ARGUMENTS**        ***character-string***  
                          VMS usage: **char\_string**  
                          type:        **character string**  
                          access:     **read only**  
                          mechanism: **by descriptor**

String whose initial character is to be used by LIB\$SKPC in the comparison. The **character-string** argument contains the address of a descriptor pointing to this string. Only the first character of **character-string** is used, and the length of **character-string** is not checked.

***source-string***  
VMS usage: **char\_string**  
type:        **character string**  
access:     **read only**  
mechanism: **by descriptor**

String to be searched by LIB\$SKPC. The **source-string** argument contains the address of a descriptor pointing to this string.

---

**DESCRIPTION**        LIB\$SKPC compares the initial character of **character-string** with successive characters of **source-string** until it finds an inequality or reaches the end of the **source-string**. It returns the relative position of this unequal character as an index, which is the relative position of the first occurrence of a substring in the source string.

---

**CONDITION**        *None.*  
**VALUES**  
**RETURNED**

---

**EXAMPLE**

```
C+
C This FORTRAN example program illustrates the use of LIB$SKPC.
C LIB$SKPC makes the VAX SKPC instruction available as a callable routine.
C LIB$SKPC compares each character of a given string with a given character.
C It returns the relative position of the first nonequal character as an index.
C-
  I = LIB$SKPC (' ', ' ABC')
  TYPE 1, I
1 FORMAT(' The blank character matches the',I2,'nd character in')
  TYPE *,'the string " ABC"'
  J = LIB$SKPC ('A', 'AAA')
  TYPE 2, J
2 FORMAT(' The character "A" matches the',I2,'th character in')
  TYPE *,'the string " AAA"'
END
```

This FORTRAN example generates the following output:

```
    The blank character matches the 2nd character in
    the string " ABC"
    The character "A" matches the 0th character in
    the string " AAA"
```



---

**DESCRIPTION** LIB\$SPANC uses successive bytes of the string specified by **source-string** to index into a table. An AND operation is performed on the byte selected from the table and the mask byte.

The operation is terminated when the result of the AND operation is zero.

---

**CONDITION**        *None.*  
**VALUES**  
**RETURNED**

---

### EXAMPLE

```

!+
! This FORTRAN program demonstrates how to use
! LIB$SCANC and STR$UPCASE.
!
! Declare the Run-Time Library routines to be used.
!-

      INTEGER*4 STR$UPCASE      ! Translate to upper case
      INTEGER*4 LIB$SCANC      ! Look for characters
      INTEGER*4 LIB$SPANC      ! Skip over characters

!+
! Declare the alphabet from which "words" are constructed.
!-

      CHARACTER*(38) ALPHABET
      DATA ALPHABET /'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789$_' /

!+
! Local variable declarations
!-

      INTEGER*4 WORD_COUNT /0/      ! Count of words found
      INTEGER*4 WORD_LENGTH /0/      ! Length of a word
      INTEGER*4 TOTAL_LENGTH /0/      ! Sum of word lengths
      INTEGER*4 START_POS /0/        ! Position of start of word
      INTEGER*4 END_POS /0/          ! Position of end of word
      REAL*4 AVERAGE_LENGTH /0.0/    ! Average length of words
      CHARACTER*80 LINE              ! Line to examine for words
      BYTE MATCH_TABLE(0:255) /256*0/ ! Match table for scanning

!+
! The routines LIB$SCANC and LIB$SPANC require a table with an entry
! for each possible character. Create a match table from ALPHABET
! with an entry of 1 if the character is in ALPHABET, 0 otherwise.
! MATCH_TABLE has already been initialized to zeros.
!-

      DO I = 1, LEN(ALPHABET)
      MATCH_TABLE(ICHAR(ALPHABET(I:I))) = 1
      END DO

!+
! Loop forever finding words in LINE. When LINE is exhausted,
! indicated by a START_POS of zero, read another one. Upon
! end-of-file, leave the loop and print the statistics.
!-

```

# LIB\$SPANC

```
OPEN( UNIT = 1, FILE = 'TEST.DAT', TYPE = 'OLD' )
DO WHILE (.TRUE.)
  DO WHILE (START_POS .EQ. 0)    ! Get a new line
    READ (1,'(A)',END=900) LINE ! If EOF, skip to 900
    CALL STR$UPCASE (LINE,LINE) ! Convert to upper
                                ! case for matching
    START_POS = LIB$SCANC (LINE,MATCH_TABLE,1) ! Find beginning
    END DO                      ! of first word

!+
! START_POS now points to the beginning of a word. Call LIB$SPANC to
! find the first character that is not part of the word. Set
! START_POS to beginning of next word. If LIB$SPANC does not
! find a non-word character, it returns zero.
!-

  END_POS =
  1   START_POS + LIB$SPANC (LINE(START_POS:), MATCH_TABLE,1) - 1
  IF (END_POS .LT. START_POS) THEN ! Word goes to end of line
    WORD_LENGTH = (LEN(LINE) + 1) - START_POS
    START_POS = 0 ! Indicate line exhausted
  ELSE
    WORD_LENGTH = END_POS - START_POS
    START_POS =
  1   END_POS + LIB$SCANC (LINE(END_POS:),MATCH_TABLE,1) - 1
  IF (START_POS .LT. END_POS) START_POS = 0 ! No more words on line
  END IF

!+
! Update count and length statistics.
!-

  WORD_COUNT = WORD_COUNT + 1
  TOTAL_LENGTH = TOTAL_LENGTH + WORD_LENGTH
  END DO

900   CONTINUE

!+
! Compute average word length and display statistics.
!-

  IF (WORD_COUNT .NE. 0)
  1   AVERAGE_LENGTH = FLOAT(TOTAL_LENGTH) / FLOAT(WORD_COUNT)
  TYPE 901,WORD_COUNT,AVERAGE_LENGTH
901  FORMAT (1X,I10,' words found, average length was ',
  1       F4.1,' letters.')

  CLOSE (1)
  END
```

This FORTRAN program reads text from the default input unit and looks for words. A word is defined as a string containing only the characters A to Z (uppercase or lowercase), 0 to 9, and the dollar sign (\$), and underscore (—) symbols. The program reports the total number of words found and their average length.

The program uses three Run-Time Library routines: STR\$UPCASE, LIB\$SCANC, and LIB\$SPANC.

- 1 The string is converted to uppercase using STR\$UPCASE so that the search for words will ignore the case of letters.
- 2 LIB\$SCANC searches through the string for one of a set of characters, the set being specified as nonzero elements in a 256-byte table.

- 3 Similarly, LIB\$SPANC uses the VAX SPANC instruction to search through a string for a character whose table entry is not zero.

The value returned by each routine is the index into the string where the first matching (or nonmatching) character was found, or zero if no match was found.

The output generated by this FORTRAN program is as follows:

12 words found, average length was 4.2 letters.





argument is the address of a descriptor pointing to this equivalence string. If omitted, the default is the caller's SYS\$OUTPUT.

### **flags**

VMS usage: **mask\_longword**  
 type: **longword (unsigned)**  
 access: **read only**  
 mechanism: **by reference**

Flag bits that designate optional behavior. The **flags** argument is the address of an unsigned longword that contains these flag bits. By default, all flags are clear.

These flags are defined as follows:

Bit	Symbol	Meaning
0	NOWAIT	If set, the calling process continues executing in parallel with the subprocess. If clear, the calling process hibernates until the subprocess completes.
1	NOCLISYM	If set, the spawned subprocess does not inherit CLI symbols from its caller. If clear, the subprocess inherits all currently defined CLI symbols. You may want to specify NOCLISYM to help prevent commands redefined by symbol assignments from affecting the spawned commands.
2	NOLOGNAM	If set, the spawned subprocess does not inherit process logical names from its caller. If clear, the subprocess inherits all currently defined process logical names. You may want to specify NOLOGNAM to help prevent commands redefined by logical name assignments from affecting the spawned commands.
3	NOKEYPAD	If set, the keypad symbols and state are passed to the subprocess. If not set, the keypad settings are not passed to the subprocess.
4	NOTIFY	If set, a message is broadcast to SYS\$OUTPUT when the subprocess completes or aborts. If not set, no message is broadcast. This bit should not be set unless the NOWAIT bit is also set.
5	NOCONTROL	If set, no carriage-return/line-feed is prefixed to any prompt string. If not set, a carriage-return/line-feed is prefixed to any prompt string specified.

Bits 6 through 31 are reserved for future expansion and must be zero. Symbolic flag names are defined in STARLET. They are CLI\$\_NOWAIT, CLI\$\_NOCLISYM, CLI\$\_NOLOGNAM, CLI\$\_NOKEYPAD, CLI\$\_NOTIFY, and CLI\$\_NOCONTROL.

### **process-name**

VMS usage: **process\_name**  
 type: **character string**  
 access: **read only**  
 mechanism: **by descriptor**

Name defined for the subprocess. The **process-name** argument is the address of a descriptor pointing to this name string. If omitted, a unique process name

# LIB\$SPAWN

will be generated. If you supply a name and it is not unique, LIB\$SPAWN will return the condition value SS\$\_DUPLNAM.

## ***process-id***

VMS usage: **process\_id**  
type: **longword (unsigned)**  
access: **write only**  
mechanism: **by reference**

Process identification of the spawned subprocess. The **process-id** argument is the address of an unsigned longword that contains this process identification value.

This process identification value is meaningful only if the NOWAIT **flags** bit is set.

## ***completion-status***

VMS usage: **longword\_unsigned**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by value**

The final completion status of the subprocess. The **completion-status** argument is an unsigned longword containing the address of the status. LIB\$SPAWN writes the address of the final completion status of the subprocess into **completion-status**. Note that **completion-status** is updated asynchronously. Your program must ensure that the address is still valid when the longword is written.

If the NOWAIT **flags** bit is set, this value is not stored until the subprocess completes; use the **byte-integer-event-flag-num** or **AST-address** arguments to determine when the subprocess has completed.

## ***byte-integer-event-flag-num***

VMS usage: **byte\_unsigned**  
type: **byte (unsigned)**  
access: **read only**  
mechanism: **by reference**

The number of a local event flag to be set when the spawned subprocess completes. The **byte-integer-event-flag-num** argument is the address of an unsigned byte that contains this event flag number. If omitted, no event flag is set.

Specifying **byte-integer-event-flag-num** is meaningful only if the NOWAIT **flags** bit is set.

## ***AST-address***

VMS usage: **procedure**  
type: **procedure entry mask**  
access: **call without stack unwinding**  
mechanism: **by value**

Entry mask of a routine to be called by means of an AST when the subprocess completes. The **AST-address** argument is the address of this procedure entry mask.

Specifying **AST-address** is meaningful only if the NOWAIT **flags** bit is set.

***varying-AST-argument***

VMS usage: **user\_arg**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **by value**

A value to be passed to the AST routine. Typically, the **varying-AST-argument** argument is the address of a block of storage the AST routine will use.

Specifying **varying-AST-argument** is meaningful only if the **NOWAIT flags** bit is set and if **AST-address** has been specified.

***prompt-string***

VMS usage: **char\_string**  
 type: **character string**  
 access: **read only**  
 mechanism: **by descriptor**

Prompt string to use in the subprocess. The **prompt-string** argument is the address of a descriptor pointing to this prompt string. If omitted, the subprocess will use the same prompt string that the parent process uses.

***cli***

VMS usage: **char\_string**  
 type: **character string**  
 access: **read only**  
 mechanism: **by descriptor**

File specification for the command language interpreter (CLI) to be run in the subprocess. The **cli** argument is the address of this file specification string's descriptor. The CLI specified must reside in SYS\$SYSTEM with a file type of EXE, and it must be installed. No directory or file type may be specified.

If omitted, the subprocess will use the same CLI as the parent process. If specified, no context will be copied to the subprocess.

---

**DESCRIPTION**

The subprocess created by LIB\$SPAWN inherits the following attributes from the caller's environment:

- Process logical names
- Global and local CLI symbols
- Default device and directory
- Process privileges
- Process nondeductible quotas
- Current command verification setting

The subprocess does not inherit process-permanent files, nor routine or image context.

If neither **command-string** nor **input-file** is present, command input will be taken from the parent terminal. If both **command-string** and **input-file** are present, the subprocess will first execute **command-string** and then read from **input-file**. If only **command-string** is specified, the command will be executed and the subprocess will be terminated. If **input-file** is specified,

# LIB\$SPAWN

the subprocess will be terminated by either a LOGOUT command or an end-of-file.

The subprocess does not inherit process-permanent files, nor routine or image context. No LOGIN.COM file is executed.

Unless the NOWAIT **flags** bit is set, the caller's process is put into hibernation until the subprocess completes. Because the caller's process hibernates in supervisor mode, any user-mode ASTs queued for delivery to the caller will not be delivered until the caller reawakes. Control can also be restored to the caller by means of an ATTACH command or by a suitable call to LIB\$ATTACH from the subprocess.

This routine is supported for use only with the DCL command language interpreter. If used when the current CLI is MCR, the error status LIB\$\_NOCLI will be returned.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In such cases the error status LIB\$\_NOCLI is returned.

Programs depending on embedded DCL commands may not function properly when run under other command language interpreters that may be supported by future versions of VMS.

See the *VMS DCL Dictionary* for a complete description of the SPAWN command.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
SS\$_ACCVIO	Access violation. One of the string arguments to LIB\$SPAWN could not be read, or <b>completion-status</b> could not be written.
SS\$_DUPLNAM	Duplicate process name. If the argument <b>process-name</b> was specified, it duplicated an existing process name. If <b>process-name</b> was omitted, LIB\$SPAWN was unable to create a unique name for the subprocess.
fac\$_xxx	Other error trying to create subprocess.
LIB\$_INVARG	Invalid argument. The optional argument <b>flags</b> was specified and a bit other than bits 0 through 5 was set.
LIB\$_INVSTRDES	Invalid string descriptor. One of the string arguments had an invalid descriptor.
LIB\$_NOCLI	No CLI present to perform function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.

If an error is encountered while trying to create the subprocess, the status value for that error is returned by LIB\$SPAWN.

---

## EXAMPLE

```
ISTAT=LIB$SPAWN(,,CLIM_NOKEYPAD.....'> ')  
IF (.NOT. ISTAT) CALL LIB$STOP(%VAL(ISTAT))
```

This FORTRAN fragment illustrates a call to LIB\$SPAWN from within a FORTRAN program. A subprocess is spawned taking input from SYS\$INPUT and giving output to SYS\$OUTPUT. The keypad state is not passed to the subprocess. A prompt string of "> " is specified for the subprocess.



# LIB\$STAT\_TIMER

See the *VMS System Services Reference Manual* for more details on the system time format.

## **handle-address**

VMS usage: **address**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Pointer to a block of storage. The optional **handle-address** argument contains the address of an unsigned longword that is this pointer.

If **handle-address** is specified, LIB\$STAT\_TIMER assumes that LIB\$INIT\_TIMER has been called with the same value of **handle-address**. **Handle-address** is an optional argument. If it is not specified, LIB\$STAT\_TIMER uses internal storage.

---

## DESCRIPTION

Only one of the five statistics is returned by each call to LIB\$STAT\_TIMER. The elapsed time is returned in the system quadword format. Therefore the receiving area should be eight bytes long. All other returned values are longwords.

LIB\$SHOW\_TIMER and LIB\$STAT\_TIMER are relatively simple tools for testing the performance of a new application. Note that LIB\$INIT\_TIMER must be called prior to any calls to LIB\$SHOW\_TIMER or LIB\$STAT\_TIMER.

To obtain more detailed information, use LIB\$GETJPI (Get Job/Process Information) or the VMS system service SYS\$GETTIM (Get Time).

The following summary illustrates the differences between LIB\$SHOW\_TIMER and LIB\$STAT\_TIMER.

---

Code	Statistic	Format for LIB\$SHOW_TIMER	Format for LIB\$STAT_TIMER
1	Elapsed real time	hhhh:mm:ss.cc	Quadword in system time format
2	Elapsed CPU time	hhhh:mm:ss.cc	Longword in 10-millisecond increments
3	Count of buffered I/O operations	nnnn	Longword
4	Count of direct I/O operations	nnnn	Longword
5	Count of page faults	nnnn	Longword

---

When you call LIB\$INIT\_TIMER, you must use the optional **handle-address** argument only if you want to keep several sets of statistics simultaneously. This argument points to a block in heap storage where the statistics are to be stored.

You need to call LIB\$FREE\_TIMER only if you have specified **handle-address** in LIB\$INIT\_TIMER and you wish to deallocate all heap storage resources. In most cases, the implicit deallocation at program exit time will be sufficient.



# LIB\$STAT\_TIMER

---

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Routine successfully completed.

LIB\$\_INVARG

Invalid argument. Either **code** or **handle-address** is invalid.

---

## EXAMPLE

```
PROGRAM STAT_TIMER(INPUT,OUTPUT);
```

```
{+}
```

```
{ This Pascal example program demonstrates the use of
```

```
{ LIB$STAT_TIMER.
```

```
{-}
```

```
TYPE
```

```
  BYTE = [BYTE] 0..255;  
  WORD = [WORD] 0..65535;  
  QUADWORD_SYSTEM_TIME = [QUAD] RECORD  
    FIRST_LONGWORD : UNSIGNED;  
    SECOND_LONGWORD : UNSIGNED;  
  END;
```

```
VAR
```

```
  ELAPSED_REAL_TIME : QUADWORD_SYSTEM_TIME;  
  ELAPSED_STRING : VARYING [32] OF CHAR;  
  PAGE_FAULT_COUNT : UNSIGNED;  
  RETURNED_STATUS : UNSIGNED;
```

```
[EXTERNAL] FUNCTION LIB$INIT_TIMER(  
  HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0  
  ) : INTEGER; EXTERNAL;
```

```
[EXTERNAL] FUNCTION LIB$STAT_TIMER(  
  CODE : INTEGER;  
  VALUE : [UNSAFE,REFERENCE] PACKED ARRAY [L..U:INTEGER] OF BYTE;  
  HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0  
  ) : INTEGER; EXTERNAL;
```

```
[EXTERNAL] FUNCTION LIB$STOP(  
  CONDITION_STATUS : [IMMEDIATE,UNSAFE] UNSIGNED;  
  FAO_ARGS : [IMMEDIATE,UNSAFE,LIST] UNSIGNED  
  ) : INTEGER; EXTERNAL;
```

```
[EXTERNAL] FUNCTION LIB$SYS_ASCTIM(  
  OUT_LEN : [REFERENCE] WORD := %IMMED 0;  
  VAR DST_STR : PACKED ARRAY [L..U:INTEGER] OF CHAR;  
  USER_TIME : QUADWORD_SYSTEM_TIME := %IMMED 0;  
  CNV_FLG : UNSIGNED := %IMMED 0  
  ) : INTEGER; EXTERNAL;
```

```
BEGIN
```

```
{+}
```

```
{ Call LIB$INIT_TIMER to initialize RTL internal counters.
```

```
{-}
```

```
RETURNED_STATUS := LIB$INIT_TIMER;
```

```
IF NOT ODD(RETURNED_STATUS)
```

```
THEN
```

```
  LIB$STOP(RETURNED_STATUS);
```

# LIB\$STAT\_TIMER

```
{+}
{ Print a line of text to waste time.
{-}

WRITELN('Spend time to acquire elapsed real time and page faults');

{+}
{ Call LIB$STAT_TIMER to retrieve statistics values.
{-}

RETURNED_STATUS := LIB$STAT_TIMER(1,ELAPSED_REAL_TIME);
IF NOT ODD(RETURNED_STATUS)
THEN
    LIB$STOP(RETURNED_STATUS);

RETURNED_STATUS := LIB$STAT_TIMER(5,PAGE_FAULT_COUNT);
IF NOT ODD(RETURNED_STATUS)
THEN
    LIB$STOP(RETURNED_STATUS);

{+}
{ Print the statistics retrieved from LIB$STAT_TIMER.
{-}

WRITELN('Page fault count is ',PAGE_FAULT_COUNT:1);

RETURNED_STATUS := LIB$SYS_ASCTIM(
    ELAPSED_STRING.LENGTH,
    ELAPSED_STRING.BODY,
    ELAPSED_REAL_TIME,
    1);
IF NOT ODD(RETURNED_STATUS)
THEN
    LIB$STOP(RETURNED_STATUS);

WRITELN('Elapsed real time is ',ELAPSED_STRING);

END.
```

This Pascal program demonstrates the use of LIB\$STAT\_TIMER. The output generated by this program is as follows:

```
Spend time to acquire elapsed real time and page faults
Page fault count is 22
Elapsed real time is 00:00:00.61
```

# LIB\$STAT\_VM

---

## LIB\$STAT\_VM Return Virtual Memory Statistics

The Return Virtual Memory Statistics routine returns to its caller one of six statistics available from calls to LIB\$GET\_VM/LIB\$FREE\_VM and LIB\$GET\_VM\_PAGE/LIB\$FREE\_VM\_PAGE. Unlike LIB\$SHOW\_VM, which formats the values for output and displays them on SY\$OUTPUT, LIB\$STAT\_VM returns the statistic in the **value-argument** argument. Only one of the statistics is returned by each call to LIB\$STAT\_VM.

---

**FORMAT**            **LIB\$STAT\_VM** *code ,value-argument*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:         **write only**  
                          mechanism:      **by value**

---

**ARGUMENTS**         **code**  
                          VMS usage: **longword\_signed**  
                          type:            **longword integer (signed)**  
                          access:         **read only**  
                          mechanism:      **by reference**

Code specifying which statistic is to be returned. The **code** argument contains the address of a signed longword integer that is this code.

---

Code	Statistic
1	Number of successful calls to LIB\$GET_VM
2	Number of successful calls to LIB\$FREE_VM
3	Number of bytes allocated by LIB\$GET_VM but not yet deallocated by LIB\$FREE_VM
5	Number of calls to LIB\$GET_VM_PAGE
6	Number of calls to LIB\$FREE_VM_PAGE
7	Number of pages allocated by LIB\$GET_VM_PAGE but not yet deallocated by LIB\$FREE_VM_PAGE

---

Note that it is invalid to omit **code** or to give a **code** of 0 or 4.

**value-argument**  
VMS usage: **user\_arg**  
type:        **unspecified**  
access:     **write only**  
mechanism: **by reference**

Value of the statistic returned by LIB\$STAT\_VM. The **value-argument** argument contains the address of a signed longword integer that is this value.

---

**DESCRIPTION**

LIB\$STAT\_VM returns to its caller one of six available statistics. Unlike LIB\$SHOW\_VM, which formats the values for output, LIB\$STAT\_VM returns the value to a location specified as an argument.

Only one of the six statistics can be returned by one call to LIB\$STAT\_VM. **Code** must be one of six values described for LIB\$SHOW\_VM. A **code** value of 0 or 4 is invalid.

Unlike LIB\$SHOW\_VM, which produces ASCII values for output, LIB\$STAT\_VM returns the value in binary form to a location specified as an argument.

---

**CONDITION  
VALUES  
RETURNED**

SS\$\_NORMAL  
LIB\$\_INVARG

Routine successfully completed.  
Invalid argument. The value of **code** was not one of the values allowed by LIB\$STAT\_VM.



***FAO-argument1***

VMS usage: **varying\_arg**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **unspecified**

Optional FAO (formatted ASCII output) argument that is associated with the specified condition value.

Section 4.1.5 explains the message format.

***condition-value2***

VMS usage: **cond\_value**  
 type: **longword (unsigned)**  
 access: **read only**  
 mechanism: **by value**

VAX 32-bit condition value. The optional **condition-value2** argument is an unsigned longword that contains this condition value.

Section 4.1.2 explains the format of a condition value.

***number-of-arguments2***

VMS usage: **longword\_signed**  
 type: **longword integer (signed)**  
 access: **read only**  
 mechanism: **by value**

Optional FAO argument associated with the condition value. The **number-of-arguments2** argument is a signed longword integer that contains this number. If omitted or specified as zero, no FAO arguments follow.

***FAO-argument2***

VMS usage: **varying\_arg**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **unspecified**

FAO (formatted ASCII output) argument that is associated with the specified condition value.

Section 4.1.5 explains the message format.

---

**DESCRIPTION**

LIB\$STOP is called whenever your program must indicate an exception condition because it is impossible to continue execution or return a status code to the calling program.

LIB\$STOP scans the stack frame by frame, starting with the most recent frame, calling each established handler (see Section 4.1.3). LIB\$STOP guarantees that control will not return to the caller.

The LIB\$STOP argument list, the Program Counter (PC) and Processor Status Longword (PSL) of the caller are appended to build the signal argument vector.

The severity of **condition-value** is forced to SEVERE before each call to a handler.

# LIB\$STOP

If any handler attempts to continue by returning a success completion code, the error message ATTEMPT TO CONTINUE FROM STOP is printed and your program exits.

If the handler called by LIB\$STOP in turn calls SYS\$UNWIND, control will not return to LIB\$STOP's caller, thus changing the program flow. A handler can also modify the saved copy of R0/R1 in the mechanism vector, changing registers R0 and R1 after the stack has been unwound. If a handler does neither of these things, then all registers including R0/R1 and the hardware condition codes are preserved.

The only way a handler can prevent the image from exiting after a call to LIB\$STOP is to unwind the stack using the SYS\$UNWIND system service.

---

## CONDITION VALUES RETURNED

*None.*

---

## EXAMPLE

```
10 EXTERNAL LONG FUNCTION LIB$RESERVE_EF
   DECLARE LONG RET_STATUS

   RET_STATUS = LIB$RESERVE_EF( 2% )
   IF (RET_STATUS AND 1%) = 0% THEN
CALL LIB$STOP( RET_STATUS BY VALUE )
   END IF

   PRINT "Event flag 2 reserved successfully"

END
```

This BASIC example program uses LIB\$STOP to stop executing if an error is signaled. This BASIC program tries to reserve an event flag that is not accessible to user programs, thus ensuring that an error will be signaled.

The output generated by this BASIC program is as follows:

```
%LIB-F-EF_ALRRES, event flag already reserved
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name      routine name      line      rel PC      abs PC
2822XBLST$MAIN   2822XBLST$MAIN   6         00000044    00000644
```

---

## LIB\$SUB\_TIMES Subtract Two Quadword Times

The Subtract Two Quadword Times routine subtracts two VMS internal-time-format times.

---

**FORMAT**            **LIB\$SUB\_TIMES**    *time1 ,time2 ,resultant-time*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENTS**        ***time1***  
                           VMS usage: **date\_time**  
                           type:        **quadword (unsigned)**  
                           access:      **read only**  
                           mechanism: **by reference**

First time, from which LIB\$SUB\_TIMES subtracts the second time. The **time1** argument is the address of an unsigned quadword containing this time. **Time1** must represent a later time or a longer time interval than **time2**. **Time1** may be either absolute time or delta time as long as **time2** is of the same type. If **time1** and **time2** are of different types, **time1** must be the absolute time.

***time2***  
                           VMS usage: **date\_time**  
                           type:        **quadword (unsigned)**  
                           access:      **read only**  
                           mechanism: **by reference**

Second time, which LIB\$SUB\_TIMES subtracts from the first time. The **time2** argument is the address of an unsigned quadword containing this time. **Time2** must represent an earlier time or a shorter time interval than **time1**. **Time2** may be either absolute time or delta time as long as **time1** is of the same type. If **time2** and **time1** are of different types, **time2** must be the delta time.

***resultant-time***  
                           VMS usage: **date\_time**  
                           type:        **quadword (unsigned)**  
                           access:      **write only**  
                           mechanism: **by reference**

The result of subtracting **time2** from **time1**. The **resultant-time** argument is the address of an unsigned quadword containing the result. If both **time1** and **time2** are delta times, then **resultant-time** is a delta time. If both **time1** and **time2** are absolute times, then **resultant-time** is a delta time. If **time1** is an absolute time and **time2** is a delta time, then **resultant-time** is an absolute time.



# LIB\$SUB\_TIMES

---

## DESCRIPTION

LIB\$SUB\_TIMES subtracts two VMS internal times. The second time, specified by **time2**, is subtracted from **time1**. The following table shows the only combinations of times you can subtract:

Time1	Time2	Subtraction	Resultant-Time
delta	delta	time1—time2	delta
absolute	absolute	time1—time2	delta
absolute	delta	time1—time2	absolute

Delta times must be less than 10,000 days.

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Normal successful completion.
LIB\$_IVTIME	Invalid time.
LIB\$_NEGTIM	Negative time computed.
LIB\$_WRONUMARG	Incorrect number of arguments.
LIB\$_INVARGORD	Invalid ordering of arguments.



# LIB\$SUBX

is this length. **Array-length** must not be negative. The default length is 2 units.

---

## DESCRIPTION

LIB\$SUBX performs subtraction on signed two's complement integers of arbitrary length. The integers are located in arrays of longwords. The higher addresses contain the higher-precision parts of the values. The highest-addressed longword contains the sign and 31 bits of precision. The remaining longwords contain 32 bits of precision in each. The number of longwords to be operated on is given by the optional argument, **array-length**. The default length is 2, which corresponds to the VAX quadword data type.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
SS\$_INTOVF	Integer overflow. The result is correct, except that the sign bit is lost.
LIB\$_INVARG	Invalid argument. Length is negative. The output array is unchanged.

---

## EXAMPLE

```
C+
C This FORTRAN example program demonstrates the use of LIB$SUBX.
C-
      INTEGER A(2),B(2),C(2),RETURN
C+
C Let "A" have the value 72057594037927937 = '1000000000000001'x.
C Let "B" have the value 4294967295      = '00000000FFFFFFFF'x.
C-
      A(1) = '00000001'x
      A(2) = '10000000'x
      B(1) = 'FFFFFFFF'x
      B(2) = '00000000'x
C+
C Then "A" - "B" is 72057589742960642.
C-
      RETURN = LIB$SUBX(A,B,C)
      TYPE *, '
      TYPE *, 'Let A = 72057594037927937 and B = 4294967295.'
      TYPE *, 'Then C = A - B = 72057589742960642.'
      TYPE 2,C(2),C(1)
2  FORMAT(' 72057589742960642 is represented as ',1H',Z8,Z8,3H'x.)
      TYPE *, 51HThat is, C(2) = '0FFFFFFF'x and C(1) = '00000002'x.
      END
```

This FORTRAN example demonstrates how to call LIB\$SUBX. The output generated by this program is as follows:

```
Let A = 72057594037927937 and B = 4294967295.
Then C = A - B = 72057589742960642.
72057589742960642 is represented as 'FFFFFFF      2'x.
That is, C(2) = '0FFFFFFF'x and C(1) = '00000002'x.
```



# LIB\$SYS\_ASCTIM

If zero or no address is specified, the current system date and time are returned. A positive value represents an absolute time. A negative value represents a delta time. Delta times must be less than 10,000 days.

## **flags**

VMS usage: **mask\_longword**  
type: **longword (unsigned)**  
access: **read only**  
mechanism: **by reference**

Conversion indicator specifying which date and time fields LIB\$SYS\_ASCTIM should return. The **flags** argument is the address of an unsigned bit mask that contains this conversion indicator.

A value of 1 causes only the hour, minute, second, and hundredths of a second to be returned, depending on the length of the buffer. A value of zero (the default) causes the full date and time to be returned, depending on the length of the buffer.

The results of specifying some possible combinations for the values of the **flags** and **time-string** arguments are shown below:

<b>Time Value</b>	<b>Time-string Length</b>	<b>Flags Value</b>	<b>Information Returned</b>
Absolute	23	0	Date and time
Absolute	12	0	Date
Absolute	11	1	Time
Delta	16	0	Days and time
Delta	11	1	Time

Argument **flags** is passed to LIB\$SYS\_ASCTIM by reference and is changed to value for use by \$ASCTIM.

---

**DESCRIPTION** See the *VMS System Services Reference Manual* for a complete description of \$ASCTIM.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Routine successfully completed, but the source string was truncated.
LIB\$_FATERRLIB	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
LIB\$_INSVIRMEM	Insufficient virtual memory. A call to LIB\$GET_VM has failed because your program has exceeded the image quota for virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.
SS\$_IVTIME	The specified delta time is greater than or equal to 10,000 days.



**directive-argument**

VMS usage: **varying\_arg**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **unspecified**

Directive argument contained in longwords. Depending on the directive, a **directive-argument** argument can be a value to be converted, the address of the string to be inserted, or a length or argument count. The passing mechanism for each of these arguments should be the one expected by the \$FAO system service.

---

**DESCRIPTION** See the *VMS System Services Reference Manual* for a complete description of \$FAO.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
SS\$_BUFFEROVF	Successfully completed, but the formatted output string overflowed the output buffer and was truncated.
LIB\$_STRTRU	Success, but the source string was truncated on copy.
SS\$_BADPARAM	An invalid directive was specified in the FAO control string.
LIB\$_INSVIRMEM	Insufficient virtual memory to allocate dynamic string.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.



# LIB\$SYS\_FAOL

---

## LIB\$SYS\_FAOL Invoke \$FAOL System Service to Format Output

The Invoke \$FAOL System Service to Format Output routine calls the system service routine \$FAOL, returning the string in the semantics you provide. If called with other than a fixed-length string for output, the length of the resultant string is limited to 256 bytes and truncation will occur.

---

<b>FORMAT</b>	<b>LIB\$SYS_FAOL</b> <i>character-string</i> <i>,[resultant-length]</i> <i>,resultant-string</i> <i>,directive-argument-address</i> <i>....</i>
---------------	--

---

<b>RETURNS</b>	VMS usage: <b>cond_value</b> type: <b>longword (unsigned)</b> access: <b>write only</b> mechanism: <b>by value</b>
----------------	---

---

<b>ARGUMENTS</b>	<b><i>character-string</i></b> VMS usage: <b>char_string</b> type: <b>character string</b> access: <b>read only</b> mechanism: <b>by descriptor</b>
------------------	---

ASCII control string, consisting of the fixed text of the output string and FAO directives. The **character-string** argument contains the address of a descriptor pointing to this control string.

***resultant-length***  
VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **write only**  
mechanism: **by reference**

Length of the output string. The **resultant-length** argument contains the address of an unsigned word integer that is this length.

***resultant-string***  
VMS usage: **char\_string**  
type: **character string**  
access: **write only**  
mechanism: **by descriptor**

Fully formatted output string returned by LIB\$SYS\_FAOL. The **resultant-string** argument contains the address of a descriptor pointing to this output string.

**directive-argument-address**

VMS usage: **address**  
 type: **longword (unsigned)**  
 access: **read only**  
 mechanism: **unspecified**

Directive arguments. The **directive-argument-address** arguments are contained in an array of unsigned longword directive arguments. Depending on the directive, a **directive-argument-address** argument can be a value to be converted, the address of the string to be inserted, or a length or argument count. The passing mechanism for each of these arguments should be the one expected by the \$FAOL system service.

---

**DESCRIPTION** See the *VMS System Services Reference Manual* for a complete description of \$FAOL.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
SS\$_BUFFEROVF	Successfully completed, but the formatted output string overflowed the output buffer and was truncated.
LIB\$_STRTRU	Success, but the source string was truncated on copy.
SS\$_BADPARAM	An invalid directive was specified in the FAO control string.
LIB\$_INSVIRMEM	Insufficient virtual memory to allocate dynamic string.
LIB\$_INVSTRDES	Invalid string descriptor. A string descriptor has an invalid value in its DSC\$_CLASS field.



**flags**

VMS usage: **mask\_longword**  
 type: **longword (unsigned)**  
 access: **read only**  
 mechanism: **by reference**

Four flag bits for message content. The **flags** argument is the address of an unsigned longword that contains these flag bits. The default value is a longword with bits zero through 3 set to 1. The **flags** argument is passed to LIB\$SYS\_GETMSG by reference and changed to value for use by \$GETMSG.

Bit numbers, their values, and corresponding descriptions are listed below.

Bit	Value	Description
0	1	Include text of message
	0	Do not include text of message
1	1	Include message identifier
	0	Do not include message identifier
2	1	Include severity indicator
	0	Do not include severity indicator
3	1	Include facility name
	0	Do not include facility name

**unsigned-resultant-array**

VMS usage: **unspecified**  
 type: **unspecified**  
 access: **write only**  
 mechanism: **by reference, array reference**

A 4-byte array to receive message-specific information. The **unsigned-resultant-array** argument contains the address of this array.

The contents of this 4-byte array are as follows:

Byte	Contents
0	Reserved
1	Count of FAO arguments
2	User value
3	Reserved

**DESCRIPTION**

LIB\$SYS\_GETMSG calls the \$GETMSG system service and returns a message string using the semantics of the caller's string. Note that, in order to retrieve a message string for a LIB\$ facility message, you must include the file \$LIBDEF in your program.

See the *VMS System Services Reference Manual* for a more complete description of \$GETMSG.

# LIB\$SYS\_GETMSG

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
SS\$_BUFFEROVF	Successfully completed, but the resultant string overflowed the buffer provided and was truncated.
SS\$_MSGNOTFND	Successfully completed, but the message code does not have an associated message on file.
LIB\$_STRTRU	Successfully completed, but the source string was truncated.
LIB\$_FATERRLIB	Fatal internal error.
LIB\$_INSVIRMEM	Insufficient virtual memory.
LIB\$_INVSTRDES	Invalid string descriptor.

---

## LIB\$TPARSE Table-Driven Finite-State Parser

The Table-Driven Finite State Parser routine is a general-purpose, table-driven parser implemented as a finite-state automaton, with extensions that make it suitable for a wide range of applications. LIB\$TPARSE parses a string and returns a message indicating whether or not the input string is valid.

---

**FORMAT**            **LIB\$TPARSE** *argument-block ,state-table ,key-table*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:            **longword (unsigned)**  
                          access:          **write only**  
                          mechanism:      **by value**

---

**ARGUMENTS**          ***argument-block***  
                          VMS usage: **address**  
                          type:            **longword (unsigned)**  
                          access:          **modify**  
                          mechanism:      **by reference**

LIB\$TPARSE argument block. The **argument-block** argument contains the address of this argument block.

The LIB\$TPARSE argument block contains information about the state of the parse operation. It becomes the argument list presented to all action routines.

Figure LIB-6 illustrates the format of the argument block.

# LIB\$TPARSE

**Figure LIB-6 LIB\$TPARSE Argument Block**

TPA\$_COUNT:	TPA\$_COUNT0 = 8	
TPA\$_OPTIONS:	Flag bits	
TPA\$_STRINGCNT:	Length of input string	
TPA\$_STRINGPTR:	Pointer to input string	
TPA\$_TOKENCNT:	Length of current token	
TPA\$_TOKENPTR:	Pointer to current token	
TPA\$_CHAR:	Character	Unused
TPA\$_NUMBER:	Binary value of numeric token	
TPA\$_PARAM:	Argument supplied by user	

ZK-1929-84

Note that the high byte of the longword field TPA\$\_OPTIONS is TPA\$\_MOUNT. The fields of the argument block are explained in detail in the section entitled "The LIB\$TPARSE Argument Block."

### **state-table**

VMS usage: **address**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **by reference**

Starting state in the state table. The **state-table** argument is the address of this starting state.

Usually, the name appearing as the first argument of the \$INIT\_STATE macro is used.

### **key-table**

VMS usage: **address**  
 type: **unspecified**  
 access: **read only**  
 mechanism: **by reference**

Keyword table. The **key-table** argument is the address of this keyword table.

This name must be the same as that which appeared as the second argument of the \$INIT\_STATE macro. It is called with the address of an argument block, the address of a state table, and the address of a keyword table. The input string is specified as part of the argument block.

---

**DESCRIPTION**

LIB\$TPARSE analyzes an input string according to a set of states and transitions presented in a state table to determine whether the input string is valid according to the rules you have defined for the input language.

The following sections explain in detail how LIB\$TPARSE works and how to call it from both assembly and high-level languages.

- 1 "How LIB\$TPARSE Works" describes the data structures used by LIB\$TPARSE and how LIB\$TPARSE operates on them.
- 2 "Coding and Using a Simple State Table" shows you how to construct and use a simple state table.
- 3 "Using Advanced LIB\$TPARSE Features" explains how to use subexpressions, abbreviations, action routines, and other advanced features.
- 4 "State Table Object Representation" includes information of interest to the low-level language programmer, such as the state table object representation.

#### How LIB\$TPARSE Works

There are three parts to any parsing operation.

- 1 The string to be parsed. LIB\$TPARSE accepts the input string as part of an argument block that contains additional information about the state of the parse—how much of the string has not been interpreted, what the current token is, and so forth. See the section entitled "The LIB\$TPARSE Argument Block."
- 2 The set of characters from which the input string is chosen, called the *alphabet* of your language. LIB\$TPARSE recognizes the ASCII character set and provides symbolic names for the most common combinations of ASCII characters—alphabetic and alphanumeric strings, VMS symbols, numbers, and so on. See the section entitled "The Alphabet of LIB\$TPARSE."
- 3 The rules which govern how the alphabet is used—in other words, the language's syntax. You specify these rules in the state tables. (In a LIB\$TPARSE state table, each state is simply a list of the transitions to other states.) See the section entitled "The State Tables."

LIB\$TPARSE reads the input string from left to right, dividing it into a set of *tokens*—substrings to be treated as logical entities. By default, LIB\$TPARSE treats blanks as invisible separators; it takes all characters up to the next blank as a single token. You can use the TPA\$V\_BLANKS flag in the argument block to cause LIB\$TPARSE to interpret the tokens differently; see the section entitled "Blanks in the Input String."

LIB\$TPARSE evaluates the transitions in the order in which they appear in the state, which corresponds to the order in which they were written in the source program. Each token is then evaluated against each possible transition in the current state. If it does not match, LIB\$TPARSE attempts to match the next transition, until it runs out of transitions in the state.

Each transition specifies what constitutes a valid token, what state is to be entered next, whether an action routine is to be called, and whether information is to be stored in a **mask** or **mask-adr** argument. By default, the next state is the state that follows the current state in the state table. No



# LIB\$TPARSE

action routines are called, and no information is stored. LIB\$TPARSE also allows subexpression calls, which can change the order in which transitions are accepted; see the section entitled "Using Subexpressions." Action routines can also change the order in which transitions are accepted. The section entitled "Action Routines" explains how LIB\$TPARSE processes action routines.

LIB\$TPARSE reads the input string, interprets the transitions in the state table, and calls the action routines (if any) until:

- 1 It executes a transition to TPA\$\_EXIT (the string is valid) at main level (that is, while it is not processing a subexpression call). It returns with the value SS\$\_NORMAL.
- 2 A transition or action routine requests that LIB\$TPARSE consider the string invalid by specifying a transition to TPA\$\_FAIL at main level. LIB\$TPARSE returns with the value LIB\$\_SYNTAXERR or an alternate failure status returned by an action routine.
- 3 An error occurs at main level. The error can be either:
  - A syntax error. All transitions in the current state fail to match the current token. LIB\$TPARSE returns LIB\$\_SYNTAXERR or an alternate failure status returned by an action routine.
  - A state table format error. One of your state table entries is invalid. LIB\$TPARSE returns LIB\$\_INVTYPE.

LIB\$TPARSE generates no signals and establishes no condition handler; action routines can signal through LIB\$TPARSE back to the calling program.

The sections that follow describe each of these parts in more detail.

## The Alphabet of LIB\$TPARSE

LIB\$TPARSE recognizes strings made up of elements of the ASCII character set. It provides all the basic building blocks needed for constructing a grammar using the ASCII character set. There are also symbols that represent the more complex constructions found in programming and command language grammar.

Table LIB-21 illustrates the alphabet of LIB\$TPARSE.

**Table LIB-21 The Alphabet of LIB\$TPARSE**

Symbol	Character Matched
'x'	The particular ASCII character. In a state table, it is expressed by enclosing the character in single quotation marks. The character can be any member of the 8-bit ASCII code set. LIB\$TPARSE does not consider uppercase and lowercase alphabetic characters and codes with different values in bit 7 to be equivalent.
TPA\$_ANY	Any single character. (The actual matching character is placed in the TPA\$_CHAR field of the argument block.)
TPA\$_ALPHA	Any alphabetic character, which includes the DEC multinational character set.
TPA\$_DIGIT	Any numeric character, that is, 0 through 9.

Table LIB-21 (Cont.) The Alphabet of LIB\$TPARSE

Symbol	Character Matched
TPA\$_STRING	Any string of one or more alphanumeric characters, that is, uppercase or lowercase A through Z, and the numeric characters 0 through 9. The string can be any length. It is bounded on the right by the first nonalphanumeric character or by the end of the string. A descriptor of the matching string is available in the argument block.
TPA\$_SYMBOL	Any string of one or more characters of the standard VAX symbol constituent set, that is, uppercase and lowercase A through Z and all DEC multinational characters, in addition to the dollar sign (\$), and the underscore (_). The string is bounded on the right by some character not in the symbol constituent set (usually a blank), or by the end of the string.
TPA\$_BLANK	Any string of one or more blanks and/or tabs.
TPA\$_DECIMAL	Any decimal number (that is, any string of one or more digits 0 through 9) whose magnitude is less than $2^{32}$ . The binary value of the number, converted in decimal radix, is placed in the argument block.
TPA\$_OCTAL	Any octal number (that is, any string of one or more digits 0 through 7) whose magnitude is less than $2^{32}$ . The binary value of the number, converted in octal radix, is placed in the argument block.
TPA\$_HEX	Any hexadecimal number (that is, any string of one or more digits 0 through 9, A through F) whose magnitude is less than $2^{32}$ . The binary value of the number, converted in hexadecimal radix, is placed in the argument block.
TPA\$_FILESPEC	Any string of one or more characters that constitutes a valid VMS file specification. The string is bounded on the right by the first character that either is not a file specification constituent character or would cause the string to violate the syntax rules of a file specification.
TPA\$_UIC	Any string that constitutes a valid VMS numerical UIC specification, bounded by square brackets or angle brackets. The binary value of the UIC, converted in octal radix, is placed in the argument block. The wildcard character (*) is permitted in the group and/or member fields; its presence results in that field being set to its largest possible value in the binary value.

# LIB\$TPARSE

Table LIB-21 (Cont.) The Alphabet of LIB\$TPARSE

Symbol	Character Matched
TPA\$_IDENT	<p>Any string that constitutes a valid VMS identifier. Identifiers may be given as numerical UICs according to the rules for TPA\$_UIC, or as alphabetic identifier names that appear in the system's rights database. The binary value of the identifier, converted in either octal or hexadecimal radix or by lookup in the system rights database, is placed in the argument block. Identifiers may be entered in any of the following forms:</p> <p>[n,m]                      &lt;n,m&gt; [name1,name2]            &lt;name1,name2&gt; [name]                     &lt;name&gt; name %Xhex-value (any above instance of number or name may also be *)</p>
'keyword'	<p>The string of characters enclosed in single quotation marks. A keyword can consist of one or more characters of the VAX symbol constituent set, that is, uppercase and lowercase A through Z, the numerals 0 through 9, the dollar sign (\$), and the underscore (_). Uppercase and lowercase alphabetic characters are treated as different characters. A state table can contain up to 220 keywords. The keyword is bounded on the right by a character not in the symbol constituent set, or by the end of the string. Keywords that are one character in length are expressed in the form 'x*' to distinguish them from the single-character symbol ('x'). They must be differentiated since they are not the same in operation.</p> <p>For example, in the input string AB+C, the single character 'A' would match the first character of this string, whereas the keyword 'A*' would not, since B in the string is in the symbol constituent set.</p>
TPA\$_LAMBDA	<p>The empty string (always matches). As it executes the transition, LIB\$TPARSE does not remove any characters from the input string. LAMBDA transitions are useful in getting action routines called under otherwise awkward circumstances, providing unconditional GOTOs to link portions of a state table together, and providing default actions in certain cases.</p>

Table LIB-21 (Cont.) The Alphabet of LIB\$TPARSE

Symbol	Character Matched
TPA\$_EOS	The end of the input string.
label	A subexpression. LIB\$TPARSE enters the state table at the indicated label and executes state transitions until a final state is entered. If the subexpression fails (that is, if it encounters a syntax error in the input string), the input string is backed up to the point at which the subroutine started, and the subexpression simply fails to match. The subexpression facility permits complex syntactic constructs that appear in many places in grammar to appear only once in the state table. It also permits a degree of nondeterministic or pushdown parsing with a parser that is otherwise deterministic and finite-state. See the section entitled "Using Subexpressions."

A theoretical finite-state machine simultaneously compares the symbol types given by all of the transitions out of a particular state with the current token. The machine then executes the one transition whose symbol type matches. Since an ordinary sequential computer executes LIB\$TPARSE, it evaluates the transitions sequentially and executes the first transition whose symbol type matches. Note also that the set of symbol types implemented by LIB\$TPARSE matches overlapping sets of tokens. For example, the token 123 could match TPA\$\_DECIMAL, TPA\$\_OCTAL, TPA\$\_STRING, or one of several other symbol types.

Thus if there is more than one transition out of a state whose symbol types match overlapping sets of tokens, you must order the symbol types carefully. For example, the TPA\$\_SYMBOL symbol type matches all keyword strings. In general, LIB\$TPARSE will never execute keyword transitions appearing in a state following a TPA\$\_SYMBOL. It is best, therefore, to order transitions of different types in order of increasing generality, as follows:

```
'keyword'
'x'
TPA$_EOS
TPA$_ALPHA
TPA$_DIGIT
TPA$_BLANK
TPA$_OCTAL
TPA$_DECIMAL
TPA$_HEX
TPA$_STRING
TPA$_SYMBOL
TPA$_UIC
TPA$_IDENT
TPA$_FILESPEC
TPA$_ANY
TPA$_LAMBDA
```

Note that subexpressions are not in this list; their placement depends on the symbol types recognized within the subexpression. If you use action routines to reject certain transitions, you can change the order in which that symbol type is placed in this order. In any case, however, LIB\$TPARSE will execute

# LIB\$TPARSE

the first transition listed in a state that is permitted to match the leftmost portion of the input string.

## The LIB\$TPARSE Argument Block

LIB\$TPARSE finds the input string through the argument block. This argument block is the impure data base upon which LIB\$TPARSE operates. That is, it is a set of variable data that can be written as well as read. It contains information about the string to be parsed, option flags for LIB\$TPARSE, and data about the current token. When LIB\$TPARSE calls an action routine, the argument block becomes the argument list of the action routine, allowing efficient reference by the routine.

The fields in the argument block have symbolic names. Assembly language programs can define these names by invoking the macro \$TPADEF (automatically loaded from the system macro library). The field names define the byte offset of the field from the start of the argument block, with the exception of the bit fields (\$V\_names), which are defined as bit offsets from the start of the containing field. In addition, bit mask values (\$M\_names) are available for the bit fields.

The same field names are available to BLISS programs from the system macro library SYS\$LIBRARY:STARLET.L32. Each name (except for the \$M\_names) is defined as a fixed-reference macro that operates on a byte-based block. The \$M\_names are defined as literals.

Table LIB-22 contains the fields of the argument block.

**Table LIB-22 Argument Block Fields**

Symbol	Meaning
TPA\$_COUNT	A longword containing the number of longwords that make up the rest of the argument block. This longword functions as the argument count when the argument block becomes the argument list to an action routine. This field must contain the value TPA\$_COUNT0 (whose numeric value is 8).
TPA\$_OPTIONS	A longword containing various option and flag bits. The defined flags are as follows:  TPA\$_BLANKS — Setting this bit causes LIB\$TPARSE to process blanks and tabs explicitly, rather than treating them as invisible separators (see the section entitled “Blanks in the Input String” on blank processing). TPA\$_ABBRFM — Setting this bit allows keywords to be abbreviated to any length. If an abbreviated keyword string is ambiguous, the first eligible transition listed in the state matches it. TPA\$_ABBREV — Setting this bit allows keywords to be abbreviated to the shortest length that is unambiguous in that state. (See the section entitled “Abbreviating Keywords” on keyword abbreviation.) TPA\$_AMBIG — LIB\$TPARSE sets this bit when it has detected an ambiguous keyword string in the current state.
TPA\$_MCOUNT	A byte containing the minimum number of characters in the abbreviation of a keyword. If zero, abbreviations are not allowed. Preventing ambiguity is the responsibility of the state table designer. If TPA\$_ABBRFM or TPA\$_ABBREV is set, LIB\$TPARSE ignores this value.
TPA\$_STRINGCNT	A longword containing the number of characters remaining in the input string.

Table LIB–22 (Cont.) Argument Block Fields

Symbol	Meaning
TPA\$_STRINGPTR	A longword containing the address of the remainder of the string being parsed. Taken together, TPA\$_STRINGPTR and TPA\$_STRINGCNT form a descriptor for the input string. Your program initializes this descriptor with the string to be parsed. When LIB\$TPARSE calls an action routine, this descriptor describes the remainder of the input string. When LIB\$TPARSE returns, this descriptor describes the portion of the input string that LIB\$TPARSE did not process. (This occurs whether LIB\$TPARSE returns success or failure.)
TPA\$_TOKENCNT	A longword containing the number of characters in the current token.
TPA\$_TOKENPTR	A longword containing the address of the current token. Taken together, TPA\$_TOKENPTR and TPA\$_TOKENCNT form a descriptor for the current token. If LIB\$TPARSE encounters a syntax error (fails to match a transition), then this descriptor describes whatever portion of the current input string would have been matched by a TPA\$_SYMBOL symbol type. If none would have matched, it describes the first remaining character in the input string. A transition to TPA\$_FAIL leaves the descriptor describing the token matched by that transition—that is, the string that failed.
TPA\$_CHAR	A byte containing the character matched by a single-character symbol type ('x', TPA\$_ANY, TPA\$_ALPHA, or TPA\$_DIGIT). The remainder of the longword is not used.
TPA\$_NUMBER	A longword containing the binary value of a numeric token (TPA\$_DECIMAL, TPA\$_OCTAL, or TPA\$_HEX), converted in the appropriate radix.
TPA\$_PARAM	A longword containing the 32-bit argument supplied by the state transition. LIB\$TPARSE modifies the three preceding fields (TPA\$_CHAR, TPA\$_NUMBER, and TPA\$_PARAM) only when it is about to call an action routine from a transition of the relevant type (or containing an explicit argument). While LIB\$TPARSE is executing transitions of unrelated types, it does not modify the fields.
TPA\$_LENGTHO	This symbol represents the number of bytes in the basic LIB\$TPARSE argument block. You must pass an argument block of at least this length (containing a count field of TPA\$_COUNT0 in TPA\$_COUNT) as the first argument to LIB\$TPARSE. You may pass a longer block if you wish to pass extra context to action routines in a modular way.

The names TPA\$\_M\_BLANKS, TPA\$\_M\_ABBRFM, TPA\$\_M\_ABBREV, and TPA\$\_M\_AMBIG define bitmasks that correspond to the location of the corresponding \$V\_ fields in the options longword.

### The State Tables

This section describes the set of macros used to construct state tables. The section entitled “Coding and Using a Simple State Table” explains how to use these macros to construct a state table.

The state table must be set up using either MACRO or BLISS. Everything else, including the action routines, can be coded in the language of your choice. Simply compile the state table separately, then link it with your program.

# LIB\$TPARSE

## MACRO State Table Generation Macro Calls

The VMS system macro library contains a set of assembler macros that allow convenient and readable coding of a LIB\$TPARSE state table. Macros exist to initialize the LIB\$TPARSE macro system, define the states in the state table, and define the transitions to other states within each state. These macros generate symbol definitions and tables. They do not produce any executable code or routine calls.

### **\$INIT\_STATE—Initialize the LIB\$TPARSE Macros**

The \$INIT\_STATE macro declares the beginning of a state table. It initializes the internals of the table generator macros and declares the locations of the state table and the keyword table. The state table is the structure containing the definitions of the states and the transitions between them. The keyword table contains the text of the keywords used in the state table.

```
$INIT_STATE      state-table ,key-table
```

#### **state-table**

The name assigned to the state table. LIB\$TPARSE equates this label to the start of the first state in the state table.

#### **key-table**

The name assigned to the keyword table. LIB\$TPARSE equates this label to the start of the keyword table.

You must supply both the address of the state table and the address of the keyword table in the call to LIB\$TPARSE to perform a parse. The \$INIT\_STATE macro can appear more than once in a program. Each occurrence defines a separate state table. No part of any state table can refer to part of any other state table.

### **\$STATE—Defining a State**

The \$STATE macro declares the beginning of a state.

```
$STATE      [label]
```

#### **label**

An optional label for the state. LIB\$TPARSE equates the label, if present, to the starting address of the state.

### **\$TRAN—Defining State Transitions**

The \$TRAN macro defines a transition from the state in which it appears to some other (or to the same) state. The arguments of the macro define, among other things, the symbol type that causes the transition to be executed, the state to which to transfer, and the action routine to call, if any.

```
$TRAN      type [,label] [,action] [,mask] [,msk-adr] [,argument]
```

#### **type**

The symbol type recognized by this transition. The transition is taken if the characters at the front of the input string match the symbol specified. The symbol can be any of the constructs discussed in the section entitled "The Alphabet of LIB\$TPARSE."

A subexpression symbol type has the syntax label.

## label

The optional target state of this transition. If present, it must be the label assigned to some state in the state table. If no label is present in the transition, LIB\$TPARSE transfers control to the next state immediately following in the state table. If the label is the expression TPA\$\_EXIT, the parsing operation in progress is terminated with a success status. If the label is the expression TPA\$\_FAIL, the parsing operation stops with a failure status, as if a syntax error had occurred.

## action

The optional address of a user-supplied action routine. If this argument is present, LIB\$TPARSE calls the named action routine before the transition is taken. The section entitled "Action Routines" describes the calling sequence of action routines and the information available to them.

Since the action routine address is self-relative, it cannot be in a shared image separate from the state table.

## mask

An optional 32-bit mask value used with the **msk-adr** argument. If the mask is present, LIB\$TPARSE performs an inclusive OR operation using this value and the longword specified by **msk-adr**. Use of the **mask** argument allows the state table to flag the fact that a certain transition was taken without the expense and overhead of calling an action routine.

## msk-adr

The optional address associated with the preceding **mask** argument. LIB\$TPARSE performs the inclusive OR operation on this address and the **mask** argument, and stores the result at the address. If the **mask** argument is present, the **msk-adr** argument must also be present.

The **msk-adr** argument can also be present without the preceding **mask** argument. In this case, it specifies an address where the routine stores information about the matching token. The information stored depends on the nature of the symbol, as follows:

- If the symbol is a number (that is, if the type code in the transition is TPA\$\_DECIMAL, TPA\$\_OCTAL, or TPA\$\_HEX), the address contains the 32-bit binary value of the number (an unsigned longword).
- If the symbol is a single character (that is, if the type code in the transition is 'x', TPA\$\_ANY, TPA\$\_ALPHA, or TPA\$\_DIGIT), the address (an unsigned byte) contains the 8-bit matching character.
- If the symbol is of any other type, the address contains the 64-bit string descriptor of the matching token (an unsigned quadword; class and data type fields in the descriptor are undefined).

Using **msk-adr** makes your program nonmodular.

The use of the **msk-adr** alone lets a parser program extract the most commonly needed information from the input string without using action routines. Note that LIB\$TPARSE stores the information, and does not perform an OR operation as it does if **mask** is present.

Since the action routine address is self-relative, it cannot be in a shared image separate from the state table.



# LIB\$TPARSE

## **argument**

An optional 32-bit value that LIB\$TPARSE passes to the action routine without interpretation. This argument can be an identifier number, an address, or any other information your action routine needs. It allows a single action routine to serve many transitions for which similar, but slightly varying, actions must be performed.

Using **argument** as an address is nonmodular.

Note that the argument appears in the state table in its absolute form. Normally, LIB\$TPARSE stores addresses as self-relative pointers; however, LIB\$TPARSE does not know the form or meaning of **argument**, so it is stored in its absolute form. If **argument** is used as an address, therefore, the resulting parsing program containing this state table will not be position-independent code (PIC).

## **\$END\_STATE—End the State Table**

The \$END\_STATE macro declares the end of the state table. It is mandatory, in order to permit the orderly cleanup of the LIB\$TPARSE macro system. The \$END\_STATE macro has no arguments. You code it as follows:

```
$END_STATE
```

## **BLISS State Table Generation Macro Calls**

The file SYS\$LIBRARY:TPAMAC.L32 contains a set of BLISS macros that allow convenient and readable coding of LIB\$TPARSE state tables in BLISS. To make the macros available to the program, include the following declaration in the module containing the state tables:

```
LIBRARY 'SYS$LIBRARY:TPAMAC';
```

BLISS requires only two macros: \$INIT\_STATE to initialize the macros and \$STATE to define each state in its entirety. The syntax of the \$INIT\_STATE and \$STATE macros are defined below.

## **\$INIT\_STATE—Initialize the LIB\$TPARSE Macros**

The \$INIT\_STATE macro initializes the LIB\$TPARSE macro system in the same manner as it does for the assembler.

```
$INIT_STATE (state-table, key-table);
```

### **state-table**

The name assigned to the state table. LIB\$TPARSE equates this label to the start of the first state in the state table.

### **key-table**

The name assigned to the keyword table. LIB\$TPARSE equates this label to the start of the keyword table.

Both names are declared as global vectors of length zero. As with the assembler macros, you can invoke \$INIT\_STATE more than once to declare several state tables within a single module.

**\$STATE—Declaring a State and Its Transitions**

In BLISS, you use the \$STATE macro to declare a state in its entirety.

```
$STATE ([label],
        ( transition ),
        ( transition ),
        ( transition )
        .
        .
        .
        );
```

**label**

Optional address of the start of the state. The compiler declares **label** as a local vector of length zero. Note that the comma following the optional label is mandatory.

**transition**

Each transition appears within the parentheses in the same form as the transition argument list for the assembler \$TRAN macro.

```
type [.label] [,action] [,mask] [,msk-adr] [,argument]
```

The arguments of each transition are expressed in exactly the same format as in the assembler macros, with the exception of the subexpression type. In BLISS, this type has the form (label).

Note that the transitions are not keyword macros. Therefore, you must use commas to indicate arguments you have skipped.

The BLISS table generation macros contain no BEGIN or END statements. This allows \$STATE macros to refer to each other. They generate all storage with OWN declarations. This means that the macros modify PSECT declarations for OWN and GLOBAL storage. Thus if other data declarations follow the state table declarations, they may not have the correct attributes. You cannot simply surround the state table with BEGIN/END, because this constitutes an expression. No declarations of any kind, including ROUTINE declarations, can follow an expression.

There are four techniques for including LIB\$TPARSE state tables in BLISS modules.

- 1 Follow the state table with explicit redeclarations of the OWN and GLOBAL PSECTs. The BLISS example in the "Examples" section uses this technique.
- 2 Place the state table in a separate module. The high-level language examples in the next section use this technique.
- 3 Place the state table between BEGIN and END statements after the declarations within a routine body.
- 4 Place the state table between BEGIN and END statements at the end of a module.

In all cases, of course, you must define all action routines, masks, addresses, and arguments with suitable declarations (which can be FORWARD or EXTERNAL). The LIB\$TPARSE macros handle the necessary FORWARD declarations for forward references to labels within the state table.

# LIB\$TPARSE

## Coding and Using a Simple State Table

LIB\$TPARSE can be used to parse programming languages, command languages, or any other grammar for which a deterministic parser is the best choice. The following sections show how to use it to parse the command language of a simple report management utility.

This hypothetical utility allows a user to perform the following activities:

- 1 Obtain a list of available reports (SHOW command).
- 2 Read reports on the terminal (READ command).
- 3 Print reports (PRINT command).
- 4 Store new reports (FILE command).

The examples use the BASIC programming language for everything except the state and keyword tables, which are coded in BLISS.

Coding a parser program using LIB\$TPARSE involves three steps:

- 1 Set up state tables to implement your language's grammar.
- 2 Define the argument block and other common variables.
- 3 Code the main program, including the call to LIB\$TPARSE.

This simple state table program does not use any action routines or other arguments. See the section entitled "Using Advanced LIB\$TPARSE Features" for information about how to use these features of LIB\$TPARSE.

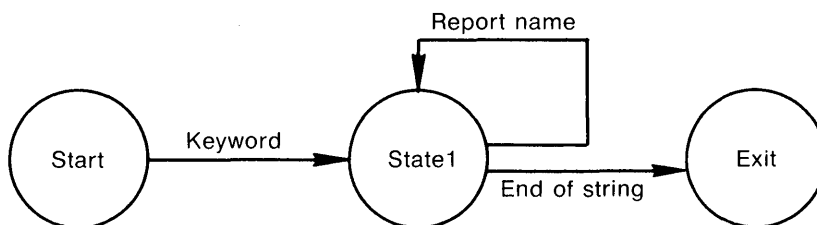
## Setting Up the State Tables

A state table associates the parser's alphabet with a set of possible transitions. You begin the state table with an \$INIT\_STATE macro and define the states and transitions using the \$STATE and \$TRAN macros.

One easy way to set up these tables is to start from a transition diagram of the language you want to parse. (If you do not know how to construct a transition diagram, you might find it helpful to read a good introductory text about compiler design and construction before you start.) Each circle in the diagram becomes a \$STATE macro, and each arrow representing a transition out of that state becomes a \$TRAN macro.

Figure LIB-7 illustrates a transition diagram for the mythical utility described in "Setting Up the State Tables".

Figure LIB-7 Transition Diagram for the Mythical Utility



ZK-1933-84

The state table for this simple language looks like this:

```

        .TITLE simplelang
        .ident 'v1'

;+
; Define the TPARSE control symbols
;-
        $TPADEF

        $INIT_STATE SIMPLE_LANGUAGE_TABLE, SIMPLE_KEYWORD_TABLE

        $STATE START
        $TRAN 'PRINT', NEED_REPORT
        $TRAN 'READ', NEED_REPORT
        $TRAN 'FILE', NEED_REPORT
        $TRAN 'SHOW', NEED_REPORT

        $STATE NEED_REPORT
        $TRAN TPA$_SYMBOL, NEED_REPORT
        $TRAN TPA$_EOS, TPA$_EXIT

        $END_STATE

        .END
  
```

Another technique for developing a state table starts with a tabular diagram in which the first column is the starting state, the second column identifies the input token, and the third gives the resultant state.

Figure LIB-8 is a diagram of the same mythical utility that appeared in LIB-7.

# LIB\$TPARSE

**Figure LIB-8 Diagram of the Mythical Utility**

Starting state	Input	Resulting state
— — — — — — — — —	PRINT READ FILE SHOW	need-report-name need-report-name need-report-name need-report-name
need-report-name need-report-name need-report-name	report name end of string other	done done error

ZK-1980-84

In this case, each block of entries becomes a \$STATE macro and each option within that block is a possible transition to the next state. Using the BLISS macros yields the following state table definition:

```

MODULE simple_statetable =
BEGIN
    !+
    ! These libraries contain the macros and other definitions
    ! needed to generate the state tables.
    !-
    LIBRARY 'SYS$LIBRARY:STARLET';
    LIBRARY 'SYS$LIBRARY:TPAMAC';
    !+
    ! UFD_STATE is the name you are giving the state table.
    ! UFD_KEY names the keyword table.
    ! Be sure to use the same name in the call to LIB$TPARSE.
    !-
    $INIT_STATE      (UFD_STATE, UFD_KEY);
    !+
    ! Read the command name (to the first blank in the command).
    ! Each string is a keyword; you are limited to 220 keywords
    ! per state table.
    !-
    $STATE (START,      ! Be careful of your punctuation here.
           ('CREATE'), ! Each transition is surrounded by
           ('FILE'),   ! parentheses; each entry except the
                       ! last is followed by a comma.
           ('PRINT'),
           ('READ')
          );
    $STATE (LOOP,
           (TPA$_STRING, LOOP), ! If there is more than one report name
                                   ! specified, go back and process it.
           (TPA$_EOS, TPA$_EXIT) ! Exit when done.
          );
END
ELUDOM                      ! End of module CREATE_TABLE

```

Assemble or compile this module as you would any other program module.

### Defining the Argument Block

After you have set up the state tables, you need to declare the LIB\$TPARSE argument block in such a way that both your program and LIB\$TPARSE can use it. This means the data must be defined in an area common to the calling program and the program module containing the state table definitions.

In most programming languages you will use a combination of EXTERNAL statements and common data definitions to create and access a separate data PSECT. If you do not know what mechanisms the language you are using provides, consult the documentation for that language.

The following example shows the LIB\$TPARSE argument block defined for use in a BASIC program.

```

!LIB$TPARSE requires that TPA$K_COUNTO be eight.
DECLARE INTEGER CONSTANT TPA$K_COUNTO = 8,      &
                    BTPA$L_COUNT = 0,          &
                    BTPA$L_OPTIONS=1,         &
                    BTPA$L_STRINGCNT=2,       &
                    BTPA$L_STRINGPTR=3,       &
                    BTPA$L_TOKENCNT=4,        &
                    BTPA$L_TOKENPTR=5,        &
                    BTPA$B_CHAR=6,            &
                    BTPA$L_NUMBER=7,          &
                    BTPA$L_PARAM=8            &

!+
! The LIB$TPARSE argument block.
!-

MAP (TPARSE_BLOCK) LONG TPARSE_ARRAY (TPA$K_COUNTO)

!+
! Redefining the map allows you to use the standard
! LIB$TPARSE symbolic names. TPA$L_STRINGCNT,
! for example, references the same storage location
! as TPARSE_ARRAY(2) and TPARSE_ARRAY(BTPA$L_STRINGCNT).
!-
MAP (TPARSE_BLOCK) LONG                                &
                    TPA$L_COUNT ,                      &
                    TPA$L_OPTIONS,                     &
                    TPA$L_STRINGCNT,                   &
                    TPA$L_STRINGPTR,                   &
                    TPA$L_TOKENCNT,                    &
                    TPA$L_TOKENPTR,                    &
                    TPA$B_CHAR,                        &
                    TPA$L_NUMBER,                      &
                    TPA$L_PARAM                        &

```

### Coding the Call to LIB\$TPARSE

Before your program can call LIB\$TPARSE, it must place the necessary information in the argument block. Since this utility uses all the LIB\$TPARSE defaults for blanks processing, abbreviations, and so on, it does not need to set any flags. It must, however, put the address and length of the string to be parsed into the TPA\$L\_STRINGCNT and TPA\$L\_STRINGPTR fields.

This information is available in the descriptor of the input string (called COMMAND\_LINE in this program). However, BASIC, like most high-level languages, does not allow you to look at the descriptors of your strings. Instead, you can use LIB\$ANALYZE\_SDESC to read the length and address from the string descriptor and place them in the argument block. (See line 75.)

# LIB\$TPARSE

Note that this program uses the BLISS state table described in the section entitled "Setting Up the State Tables."

5 %TITLE "Program to demonstrate using LIB\$TPARSE from a high-level language"

```
OPTION TYPE=EXPLICIT

!+
! COMMAND_LINE is the string to receive the input
!   command from the terminal.
! ERROR_MSG_TEXT is the system error message
!   returned from LIB$SYS_GETMSG
!   (used in the error handling routine)
!-
DECLARE STRING COMMAND_LINE, ERROR_MSG_TEXT

!+
! RET_STATUS receives the status from the system calls.
! SAVE_STATUS is used when an error occurs
!   and the error handling routine calls
!   LIB$SYS_GETMSG to obtain the error text.
!-
DECLARE LONG RET_STATUS, SAVE_STATUS

!+
! UFD_STATE is the address of the state table.
! UFD_KEY is the address of the key table.
! Both addresses are set up by the macros in module
!   SIMPLE_STATETABLE32.
!-
EXTERNAL LONG UFD_STATE, UFD_KEY

!+
! To allow us to compare returned statuses more easily.
!-
EXTERNAL INTEGER CONSTANT SS$_NORMAL,    &
                        LIB$_SYNTAXERR,  &
                        LIB$_INVTYPE

!+
! This program calls the following Run-Time Library
!   routines:
!
! LIB$TPARSE to parse the input string
!
! LIB$ANALYZE_SDESC to get the length and starting
!   address of the command string and place them
!   in the LIB$TPARSE argument block.
!
! LIB$SYS_GETMSG to find the facility, severity, and text
!   of any system errors that occur
!   during program execution.
!-
EXTERNAL LONG FUNCTION LIB$TPARSE,      &
                        LIB$ANALYZE_SDESC, &
                        LIB$SYS_GETMSG
```

# LIB\$TPARSE

```
20      !+
      ! This file defines the argument block that is passed
      ! to LIB$TPARSE. It also defines subscripts that
      ! make it easier to access the array.
      !
      ! Keeping the argument block definitions in a separate
      ! file makes them easier to modify and lets other
      ! programs use the same definitions.
      !-

      %INCLUDE "SIMPLE_TPARSE_BLOCK"

50      ON ERROR GOTO ERROR_HANDLER

60      !+
      ! LIB$TPARSE requires that TPA$L_COUNT, the
      ! first field in the argument block, have a value
      ! of TPA$K_COUNTO, whose value is 8.
      !-

      TPA$L_COUNT = TPA$K_COUNTO

75      !+
      ! Prompt at the terminal for the user's action.
      ! A real utility should provide a friendlier,
      ! clearer interface.
      !-

      GET_INPUT:      PRINT "Your options are: ", " READ report "
                     PRINT , " FILE report "
                     PRINT , " PRINT report "
                     PRINT , " CREATE report "
                     PRINT
                     INPUT "What would you like to do"; COMMAND_LINE

      !+
      ! Get the length and starting address of the command line
      ! and place them in the LIB$TPARSE argument block. Note
      ! that LIB$ANALYZE_SDESC stores the length as a word.
      !-

      RET_STATUS = LIB$ANALYZE_SDESC (COMMAND_LINE BY DESC, &
                                     TPARSE_ARRAY (BTPA$L_STRINGCNT) BY REF, &
                                     TPARSE_ARRAY (BTPA$L_STRINGPTR) BY REF)

      IF RET_STATUS <> SS$_NORMAL THEN
          GOTO ERROR_HANDLER
      END IF

100     !+
      ! Call LIB$TPARSE to process the input string.
      !
      ! Note that LIB$TPARSE expects to receive its arguments
      ! by reference, while BASIC's default for arrays and
      ! strings is by descriptor. Therefore the BY REF
      ! clauses are required. Without them, LIB$TPARSE
      ! cannot find the input string
      ! and the parse will always fail.
      !-

      RET_STATUS = LIB$TPARSE (TPARSE_ARRAY ( ) BY REF, &
                              UFD_STATE BY REF, &
                              UFD_KEY BY REF )
```



# LIB\$TPARSE

```
!+
! This simple program provides no information except that
! a valid command was entered. The next section discusses
! techniques for gathering more information.
!-

IF RET_STATUS = SS$_NORMAL

!+
! For now, exit on success.
!-

        THEN PRINT "Parse successful"
            GOTO 9999

!+
! If the parse failed, give the user a chance to try again.
!-

        ELSE IF RET_STATUS = LIB$_SYNTAXERR THEN
            PRINT "You did not enter a valid command."
            PRINT "Please try again."
            GOTO GET_INPUT

!+
! If a more serious error occurred, inform the user
! and exit.
!-

        ELSE
            Goto ERROR_HANDLER
        END IF
    END IF

500  ERROR_HANDLER: SAVE_STATUS = RET_STATUS

        RET_STATUS = LIB$_SYS_GETMSG (SAVE_STATUS, ,ERROR_MSG_TEXT)
        PRINT "Something went wrong."
        PRINT ERL, ERROR_MSG_TEXT
        RESUME 9999

9999  END
```

Compile this program as you would any other BASIC program.

When both the state tables and the main program have been compiled, link them together to form a single executable image, as follows:

```
$ LINK SIMPLANG,SIMPLANG_STATETABLE
```

## Using Advanced LIB\$TPARSE Features

The simple LIB\$TPARSE call in the previous program tells you that the command the user entered was valid, but nothing else—not even which command was entered. Most of the time your program will need more information than this.

The following sections describe some of the more complicated techniques you can use to gather extra information for your program.

## Action Routines

When the transition being matched specifies an action routine to be called, LIB\$TPARSE stores the optional argument longword, if it is present, in the argument block and calls the action routine. If the action routine returns failure, LIB\$TPARSE continues attempting to match successive transitions. If the action routine returns success, LIB\$TPARSE executes the transition as it would if there was no action routine present. It stores the mask or other value at the mask address, if specified, and passes control to the specified target state. If no target state is given, control passes to the next state following in the state table. In either case, LIB\$TPARSE does not evaluate the remaining transitions in the state.

LIB\$TPARSE calls action routines with a CALLG instruction. When a state transition specifies an action routine, LIB\$TPARSE calls the action routine when the transition is found to be able to execute successfully (that is, when its symbol type matches a leading portion of the input string). It calls the action routine before processing the **mask** or **msk-adr** arguments of the state transition.

The argument list for the action routine is the LIB\$TPARSE argument block. Thus an action routine written in assembly language, for example, can reference fields in the argument block by their symbolic offsets relative to the AP (Argument Pointer) register.

The action routine returns a value to LIB\$TPARSE in R0 that controls execution of the current state transition. If the action routine returns success (low bit set in R0), then LIB\$TPARSE proceeds with the execution of the state transition. If the action routine returns failure (low bit clear in R0), LIB\$TPARSE rejects the transition that was being processed and acts as if the symbol type of that transition had not matched. It proceeds to evaluate other transitions in that state for eligibility.

If an action routine returns a nonzero failure status to LIB\$TPARSE and no subsequent transitions in that state match, LIB\$TPARSE will return the status of the action routine, rather than the status LIB\$\_SYNTAXERR. In longword-valued functions in high-level languages, this value is returned in R0.

Allowing action routines to reject a state transition allows you to implement symbol types specific to particular applications. To recognize a specialized symbol type, code a state transition using a LIB\$TPARSE symbol type that describes a superset of the desired set of possible tokens. The associated action routine then performs the additional discrimination necessary and returns success or failure to LIB\$TPARSE, which then accordingly executes or fails to execute the transition.

A pure finite-state machine, for instance, has difficulty recognizing strings that are shorter than some maximum length, or accepting numeric values confined to some particular range.

## Blanks in the Input String

The default mode of operation in LIB\$TPARSE is to treat blanks as invisible separators. That is, they can appear between any two tokens in the string being parsed without being called for by transitions in the state table. Since blanks are significant in some situations, LIB\$TPARSE processes blanks if you have set the bit TPA\$V\_BLANKS in the options longword of the argument block. The following input string illustrates the difference in operation:

ABC DEF

# LIB\$TPARSE

LIB\$TPARSE recognizes the string by the following sequences of state transitions, depending on the state of the blanks control flag.

TPA\$V\_BLANKS set:

```
$STATE
$TRAN TPA$_STRING

$STATE
$TRAN TPA$_BLANK

$STATE
$TRAN TPA$_STRING
```

TPA\$V\_BLANKS clear:

```
$STATE
$TRAN TPA$_STRING

$STATE
$TRAN TPA$_STRING
```

Your action routines can set or clear TPA\$V\_BLANKS as LIB\$TPARSE enters or leaves sections of the state table in which blanks are significant. LIB\$TPARSE always checks the blanks control flag as it enters a state. If the flag is clear, it removes any space or tab characters present at the front of the input string before it proceeds to evaluate transitions. Note that when the TPA\$V\_BLANKS flag is clear, the TPA\$\_BLANK symbol type will never match. If TPA\$V\_BLANKS is set, you must explicitly process blanks.

## Special Characters in the Input String

Not all members of the ASCII character set can be entered directly in the state table definitions. Examples include the single quotation mark and all control characters.

In MACRO state tables, such characters can be specified as the symbol type with any assembler expression that is equivalent to the ASCII code of the desired character, not including the single quotes. For example, you could code a transition to match a backspace character as follows:

```
BACKSPACE = 8
.
.
.
$TRAN BACKSPACE, ...
```

MACRO places extra restrictions on the use of a comma in arguments to macros; often they must be surrounded by one or more angle brackets. Using a symbolic name for the comma will avoid such difficulties.

To build a transition matching such a single character in a BLISS state table, you can use the %CHAR lexical function as follows:

```
LITERAL BACKSPACE = 8;
.
.
.
$STATE (label,
        (%CHAR (BACKSPACE), ... )
        );
```

## Abbreviating Keywords

The default mode of LIB\$TPARSE is exact match. All keywords in the input string must exactly match their spelling, length, and case in the state table. However, many languages (command languages in particular) allow you to abbreviate keywords. For this reason, LIB\$TPARSE has three abbreviation facilities to permit the recognition of abbreviated keywords when the state table lists only the full spellings.

- By setting a value in TPA\$B\_MCOUNT in the LIB\$TPARSE argument block, the calling program or action routine specifies a minimum number of characters from the abbreviated keyword that must be present for a match to occur. For example, setting the byte to the value 4 would allow the keyword DEASSIGN to appear in an input string as DEAS, DEASS, DEASSI, DEASSIG, or DEASSIGN.

LIB\$TPARSE checks all the characters of the keyword string. Incorrect spellings beyond the minimum abbreviation are not permitted.

- If TPA\$V\_ABBRFM is set in the options longword, LIB\$TPARSE will recognize any leftmost substring of a keyword as a match for that keyword. LIB\$TPARSE does not check for ambiguity; it matches the first keyword listed in the state table of which the input token is a subset.
- If TPA\$V\_ABBREV is set in the options longword, LIB\$TPARSE will recognize any abbreviation of a keyword as long as it is unambiguous among the keywords in that state. If LIB\$TPARSE finds that the front of the input string contains an ambiguous keyword string, it sets the bit TPA\$V\_AMBIG in the options longword and refuses to recognize any keyword transitions in that state. (It still accepts other symbol types.) The TPA\$V\_AMBIG flag can be checked by an action routine that is called when coming out of that state, or by the calling program if LIB\$TPARSE returns with a syntax error status. LIB\$TPARSE clears the flag when it enters the next state.

For proper recognition of ambiguous keywords, the keywords in each state must be arranged in alphabetical order by the ASCII collating sequence, which is as follows:

- 1 Dollar sign (\$)
- 2 Numerics
- 3 Uppercase alphabets
- 4 Underscore (\_)
- 5 Lowercase alphabets

Be careful when using these options, since permitting short abbreviations restricts the extensibility of a language. Often, adding a new keyword can make a formerly valid abbreviation ambiguous.

If both TPA\$V\_ABBRFM and TPA\$V\_ABBREV are set, TPA\$V\_ABBRFM takes precedence.

# LIB\$TPARSE

## Using Subexpressions

LIB\$TPARSE subexpressions are analogous to subroutines within the state table. A subexpression call, indicated with the MACRO expression **!label** or the BLISS expression **(label)**, causes LIB\$TPARSE to call itself recursively, using the same argument block and keyword table, and the specified label as a starting state. LIB\$TPARSE processes the state transitions, consuming the portion of the input string called for. When LIB\$TPARSE executes a transition to TPA\$\_EXIT, it returns success to itself. LIB\$TPARSE thus considers the subexpression call a match, calls the action routine, and executes the transition. If the parse of the subexpression fails, LIB\$TPARSE returns the portion that it consumed during the failed parse to the input string, and evaluates the remaining transitions in the state.

You can use subexpressions as you would use subroutines in any program: to avoid replication of complex expressions. Subexpressions can also be used for a limited form of pushdown parsing, in which the state table contains recursively nested subexpressions. Finally, you can use subexpressions for nondeterministic parsing, that is, parsing in which you need some number of states of look-ahead. To do this, place each path of look-ahead in a separate subexpression and call the subexpressions in the transitions of the state that needs the look-ahead. When a look-ahead path fails, the subexpression failure mechanism causes LIB\$TPARSE to back out and try another path.

You should be careful when designing subexpressions that contain calls to action routines or use the **mask** and **msk-adr** transition arguments. As LIB\$TPARSE processes state transitions of a subexpression, it calls the specified action routines and stores the **mask** and **msk-adr**. If the subexpression fails, LIB\$TPARSE will back up the input string and resume processing in the calling state. However, any effects that the action routines have had on the caller's data base cannot be undone. If subexpressions are simply being used as state table subroutines, there is usually no harm done, since when a subexpression fails in this mode, the parse will generally fail. This is not true of pushdown or nondeterministic parsing. In applications where you expect subexpressions to fail, design action routines to store results in temporary storage. You can then make these results permanent at the main level, where the flow of control is deterministic.

## Using Subexpressions to Reject Transitions

The following example is an excerpt of a state table that parses a string quoted by an arbitrary character. The table interprets the first character to appear as a quote character. Many text editors and some programming languages contain this sort of construction. Executing this set of state transitions leaves a descriptor for the string in the two longwords at Q\_DESCRIPTOR, and the quoting character at location Q\_CHAR.

```

;+
; Main level state table. The first transition accepts and
; stores the quoting character.
;-
    $STATE    STRING
    $TRAN     TPA$_ANY,,,Q_CHAR
;+
; Call the subexpression to accept the quoted string and store
; the string descriptor. Note that the descriptor spans all
; the characters accepted by the subexpression.
;-
    $STATE
    $TRAN     !Q_STRING,,,Q_DESCRIPTOR
;+
; Accept the trailing quote character, left behind by the
; subexpression
;-
    $STATE
    $TRAN     TPA$_ANY,NEXT
;+
; Subexpression to scan the quoted string. The first transition
; matches until it is rejected by the action routine.
;-
    $STATE    Q_STRING
    $TRAN     TPA$_ANY,Q_STRING,TEST_Q
    $TRAN     TPA$_LAMBDA,TPA$_EXIT
;+
; The following MACRO subroutine compares the current character
; with the quoting character and returns failure if it matches.
;-
TEST_Q: .WORD    0                ; null entry mask
        CMPB    TPA$_B_CHAR(AP),Q_CHAR ; check the character
        BNEQ   10$                ; note R0 is already 1
        CLRL   R0                  ; match - reject transition
10$:    RET

```

### Using Subexpressions to Parse Complex Grammars

The following example is an excerpt from a state table that shows how to use subexpressions to parse complex grammars. The state table accepts a number followed by a keyword qualifier. Depending on the keyword, the table interprets the number as decimal, octal, or hexadecimal. The state table will accept strings such as the following:

```

10/OCTAL
32768/DECIMAL
77AF/HEX

```

This sort of grammar is difficult to parse with a deterministic finite-state machine. Using a subexpression look-ahead of two states permits a simpler expression of the state tables.

```

;+
; Main state table entry. Accept a number of some type and store
; its value at the location NUMBER.
;-
    $STATE
    $TRAN     !OCT_NUM,NEXT,,,NUMBER
    $TRAN     !DEC_NUM,NEXT,,,NUMBER
    $TRAN     !HEX_NUM,NEXT,,,NUMBER

```

# LIB\$TPARSE

```

;+
; Subexpressions to accept an octal number followed by the OCTAL
; qualifier.
;-
    $STATE    OCT_NUM
    $TRAN     TPA$_OCTAL
    $STATE
    $TRAN     '/'
    $STATE
    $TRAN     'OCTAL', TPA$_EXIT
;+
; Subexpression to accept a decimal number followed by the DECIMAL
; qualifier.
;-
    $STATE    DEC_NUM
    $TRAN     TPA$_DECIMAL
    $STATE
    $TRAN     '/'
    $STATE
    $TRAN     'DECIMAL', TPA$_EXIT
;+
; Subexpression to accept a hex number followed by the HEX
; qualifier.
;-
    $STATE    HEX_NUM
    $TRAN     TPA$_HEX
    $STATE
    $TRAN     '/'
    $STATE
    $TRAN     'HEX', TPA$_EXIT

```

Note that the transitions following the numeric token do not disturb the TPA\$\_NUMBER longword, allowing the main level subexpression call to retrieve it.

## LIB\$TPARSE and Modularity

To use LIB\$TPARSE in a modular and shareable fashion, make sure you avoid using OWN storage. Instead, allocate the argument block on the stack or the heap.

Do not use the **mask-adr** argument at all. Do not use the **argument** argument as an address.

If additional context is needed, allocate it at the end of the argument block.

You will need to use action routines to control flags such as TPA\$V\_BLANKS. The MACRO example in the Examples section illustrates such an action routine, though the program itself is not modular.

## State Table Object Representation

This section describes the binary representation of a LIB\$TPARSE state table.

Each state consists of its transitions concatenated in memory. LIB\$TPARSE equates the state label to the address of the first byte of the first transition. A marker in the last transition identifies the end of the state. The LIB\$TPARSE table macros build the state table in the PSECT \_LIB\$STATE\$.

Each transition in a state consists of 2 to 23 bytes containing the arguments of the transition. The state table generation macros do not allocate storage for arguments not specified in the transition macro. This allows simple transitions to be represented efficiently. For example, the following transition, which simply accepts the character '?' and falls through to the next state, is represented in two bytes:

```
$TRAN '?'
```

In this section, pointers described as self-relative are signed displacements from the address following the end of the pointer (this is identical to branch displacements in the VAX instruction set).

A state transition consists of the following elements:

- Symbol Type—One Byte

The first byte of a transition contains the binary coding of the symbol type accepted by this transition. It is always present. Flag bit 0 in the flags byte controls the interpretation of the type byte. If the flag is clear, then the type byte represents a single character (the 'x' construct). If the flag bit is set, then the type byte is one of the other type codes (keyword, number, and so forth). The symbol types accepted by LIB\$TPARSE are encoded as follows:

Symbol Type	Binary Encoding
'x'	ASCII code of the character (8 bits)
'keyword'	The keyword index (0 up to 219)
TPA\$_FILESPEC	234
TPA\$_UIC	235
TPA\$_IDENT	236
TPA\$_ANY	237
TPA\$_ALPHA	238
TPA\$_DIGIT	239
TPA\$_STRING	240
TPA\$_SYMBOL	241
TPA\$_BLANK	242
TPA\$_DECIMAL	243
TPA\$_OCTAL	244
TPA\$_HEX	245
TPA\$_LAMBDA	246
TPA\$_EOS	247
TPA\$_SUBEXPR	248 (subexpression call)
	(Other codes are reserved for expansion)

**Note:** Use of the symbol types TPA\$\_FILESPEC and TPA\$\_IDENT will result in calls to the VMS system services \$FILESCAN and \$ASCTOID, respectively. If your application of LIB\$TPARSE runs in an environment other than VMS user mode, you must carefully evaluate whether use of these services is consistent with your environment.



# LIB\$TPARSE

- First Flags Byte—One Byte

This byte contains the following bits, which specify the options of the transition. It is always present.

Bit	Description
0	Set if the type byte is not a single character
1	Set if the second flags byte is present
2	Set if this is the last transition in the state
3	Set if a subexpression pointer is present
4	Set if an explicit target state is present
5	Set if the <b>mask</b> longword is present
6	Set if the <b>msk-adr</b> longword is present
7	Set if an action routine address is present

- Second Flags Byte—One Byte

This byte is present if any of its flag bits is set. It contains an additional flag describing the transition. It is used as follows:

Bit 0 Set if the action routine argument is present

- Subexpression Pointer—Two Bytes

This word is present in transitions that are subexpression calls. It is a 16-bit signed self-relative pointer to the starting state of the subexpression.

- Argument Longword—Four Bytes

This longword contains the 32-bit action routine argument, when specified.

- Action Routine Address—Four Bytes

This longword contains a self-relative pointer to the action routine, when specified.

- Bit Mask—Four Bytes

This longword contains the **mask** argument, when specified.

- Mask Address—Four Bytes

This longword, when specified, contains a self-relative pointer through which the **mask**, or data that depends on the symbol type, is to be stored. Because the pointer is self-relative, when it points to an absolute location, the state table is not PIC (position-independent code).

- Transition Target—Two Bytes

This word, when specified, contains the address of the target state of the transition. The address is stored as a 16-bit signed self-relative pointer. The final state TPA\$\_EXIT is coded as a word whose value is -1; the failure state TPA\$\_FAIL is coded as a word whose value is -2.

- Keyword Table

This table is the structure to which the \$INIT\_STATE macro equates its second argument. The table is a vector of 16-bit signed pointers which address locations in the keyword string area, relative to the start of the keyword vector. As the state table source generates keywords, the LIB\$TPARSE macros assign an index number to each keyword. The index number is stored in the symbol type byte in the transition; it locates the associated keyword vector entry. The keyword strings are stored in the order encountered in the state table. Each keyword string is terminated by a byte containing the value -1. Between the keywords of adjacent states is an additional -1 byte to stop the ambiguous keyword scan.

To ensure that the keyword vector is adjacent to the keyword string area, the keyword vector is located in PSECT `_LIB$KEY0$` and the keyword strings and stored in PSECT `_LIB$KEY1$`.

Your program should not use any of the three PSECTs used by LIB\$TPARSE (`_LIB$STATE$`, `_LIB$KEY0$`, and `_LIB$KEY1$`). The PSECTs `_LIB$KEY0$` and `_LIB$KEY1$` refer to each other using 16-bit displacements, so user PSECTs inserted between them can cause truncation errors from the linker.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed. LIB\$TPARSE has executed a transition to TPA\$_EXIT at main level, not within a subexpression.
LIB\$_SYNTAXERR	Parse completed with syntax error. LIB\$TPARSE has encountered a state at main level in which none of the transitions match the input string, or in which a transition to TPA\$_FAIL was executed.
LIB\$_INVTYPE	State table error. LIB\$TPARSE has encountered an invalid entry in the state table.
Other	If an action routine returns a failure status other than zero, and the parse consequently fails, LIB\$TPARSE returns the status returned by the action routine.

---

## EXAMPLES

```

1  MODULE CREATE_DIR (                                ! Create directory file
      IDENT = 'X0000',
      MAIN = CREATE_DIR) =
BEGIN

```

# LIB\$TPARSE

```
!+
! This BLISS program accepts and parses the command line
! of a CREATE/DIRECTORY command. This program uses the
! LIB$GET_FOREIGN call to acquire the command line from
! the CLI and parse it with LIB$TPARSE, leaving the necessary
! information in its global data base. The command line is of
! the following format:
!
!     CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
!           /UIC=[2437,25]
!           /ENTRIES=100
!           /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
!
! The three qualifiers are optional. Alternatively, the command
! may take the form
!
!     CREATE/DIR DEVICE:[202,31]
!
! using any of the optional qualifiers.
!-
!+
! Global data, control blocks, etc.
!-

LIBRARY 'SYS$LIBRARY:STARLET';
LIBRARY 'SYS$LIBRARY:TPAMAC.L32';

!+
! Macro to make the TPARSE control block addressable as a block
! through the argument pointer.
!-

MACRO
    TPARSE_ARGS =
        BUILTIN AP;
        MAP AP : REF BLOCK [,BYTE];
        %;

!+
! Declare routines in this module.
!-

FORWARD ROUTINE
    CREATE_DIR,           ! Mail program
    BLANKS_OFF,          ! No explicit blank processing
    CHECK_UIC,           ! Validate and assemble UIC
    STORE_NAME,          ! Store next directory name
    MAKE_UIC;            ! Make UIC into directory name

!+
! Define parser flag bits for flags longword.
!-

LITERAL
    UIC_FLAG      = 0,      ! /UIC seen
    ENTRIES_FLAG  = 1,      ! /ENTRIES seen
    PROT_FLAG     = 2;      ! /PROTECTION seen

OWN
!+
! This is the LIB$GET_FOREIGN descriptor block to get the command line.
!-

    COMMAND_DESC      : BLOCK [DSC$K_S_BLN, BYTE],
    COMMAND_BUFF      : VECTOR [256, BYTE],
```

# LIB\$TPARSE

```
!+
! This is the TPARSE argument block.
!-

    TPARSE_BLOCK      : BLOCK [TPA$K_LENGTHO, BYTE]
                      INITIAL (TPA$K_COUNTO,      ! Longword count
                               TPA$M_ABBREV       ! Allow abbreviation
                               OR TPA$M_BLANKS),  ! Process spaces explicitly

!+
! Parser global data:
!-

    PARSE_FLAGS      : BITVECTOR [32], ! Keyword flags
    DEVICE_STRING    : VECTOR [2],     ! Device string descriptor
    ENTRY_COUNT,     ! Space to preallocate
    FILE_PROTECT,    ! Directory file protection
    UIC_GROUP,       ! Temp for UIC group
    UIC_MEMBER,      ! Temp for UIC member
    FILE_OWNER,      ! Actual file owner UIC
    NAME_COUNT,      ! Number of directory names
    UIC_STRING       : VECTOR [6, BYTE], ! Buffer for string
    NAME_VECTOR      : BLOCKVECTOR [0, 2], ! Vector of descriptors

    DIRNAME1        : VECTOR [2],     ! Name descriptor 1
    DIRNAME2        : VECTOR [2],     ! Name descriptor 2
    DIRNAME3        : VECTOR [2],     ! Name descriptor 3
    DIRNAME4        : VECTOR [2],     ! Name descriptor 4
    DIRNAME5        : VECTOR [2],     ! Name descriptor 5
    DIRNAME6        : VECTOR [2],     ! Name descriptor 6
    DIRNAME7        : VECTOR [2],     ! Name descriptor 7
    DIRNAME8        : VECTOR [2];     ! Name descriptor 8

!+
! Structure macro to reference the descriptor fields in the vector of
! descriptors.
!-

MACRO
    STRING_COUNT      = 0, 0, 32, 0%,      ! Count field
    STRING_ADDR       = 1, 0, 32, 0%;      ! Address field

!+
! TPARSE state table to parse the command line
!-

$INIT_STATE          (UFD_STATE, UFD_KEY);

!+
! Read over the command name (to the first blank in the command).
!-

$STATE (START,
        (TPA$_BLANK, , BLANKS_OFF),
        (TPA$_ANY, START)
        );

!+
! Read device name string and trailing colon.
!-

$STATE (,
        (TPA$_SYMBOL, , , , DEVICE_STRING)
        );

$STATE (,
        (':')
        );
```

# LIB\$TPARSE

```
!+
! Read directory string, which is either a UIC string or a general
! directory string.
!-

$STATE (,
        ((UIC),, MAKE_UIC),
        ((NAME))
       );

!+
! Scan for options until end of line is reached.
!-

$STATE (OPTIONS,
        ('/'),
        (TPA$_EOS, TPA$_EXIT)
       );

$STATE (,
        ('UIC', PARSE_UIC,, 1^UIC_FLAG, PARSE_FLAGS),
        ('ENTRIES', PARSE_ENTRIES,, 1^ENTRIES_FLAG, PARSE_FLAGS),
        ('PROTECTION', PARSE_PROT,, 1^PROT_FLAG, PARSE_FLAGS)
       );

!+
! Get file owner UIC.
!-

$STATE (PARSE_UIC,
        (':'),
        ('=')
       );

$STATE (,
        ((UIC), OPTIONS)
       );

!+
! Get number of directory entries.
!-

$STATE (PARSE_ENTRIES,
        (':'),
        ('=')
       );

$STATE (,
        (TPA$_DECIMAL, OPTIONS,, ENTRY_COUNT)
       );

!+
! Get directory file protection. Note that the bit masks generate the
! protection in complement form. It will be uncomplemented by the main
! program.
!-

$STATE (PARSE_PROT,
        (':'),
        ('=')
       );

$STATE (,
        ('(')
       );
```

# LIB\$TPARSE

```
$STATE (NEXT_PRO,  
       ('SYSTEM', SYPR),  
       ('OWNER', OWPR),  
       ('GROUP', GRPR),  
       ('WORLD', WOPR)  
       );  
  
$STATE (SYPR,  
       (':'),  
       ('='))  
       );  
  
$STATE (SYPRO,  
       ('R', SYPRO,, %X'0001', FILE_PROTECT),  
       ('W', SYPRO,, %X'0002', FILE_PROTECT),  
       ('E', SYPRO,, %X'0004', FILE_PROTECT),  
       ('D', SYPRO,, %X'0008', FILE_PROTECT),  
       (TPA$_LAMBDA, ENDPRO)  
       );  
  
$STATE (OWPR,  
       (':'),  
       ('='))  
       );  
  
$STATE (OWPRO,  
       ('R', OWPRO,, %X'0010', FILE_PROTECT),  
       ('W', OWPRO,, %X'0020', FILE_PROTECT),  
       ('E', OWPRO,, %X'0040', FILE_PROTECT),  
       ('D', OWPRO,, %X'0080', FILE_PROTECT),  
       (TPA$_LAMBDA, ENDPRO)  
       );  
  
$STATE (GRPR,  
       (':'),  
       ('='))  
       );  
  
$STATE (GRPRO,  
       ('R', GRPRO,, %X'0100', FILE_PROTECT),  
       ('W', GRPRO,, %X'0200', FILE_PROTECT),  
       ('E', GRPRO,, %X'0400', FILE_PROTECT),  
       ('D', GRPRO,, %X'0800', FILE_PROTECT),  
       (TPA$_LAMBDA, ENDPRO)  
       );  
  
$STATE (WOPR,  
       (':'),  
       ('='))  
       );  
  
$STATE (WOPRO,  
       ('R', WOPRO,, %X'1000', FILE_PROTECT),  
       ('W', WOPRO,, %X'2000', FILE_PROTECT),  
       ('E', WOPRO,, %X'4000', FILE_PROTECT),  
       ('D', WOPRO,, %X'8000', FILE_PROTECT),  
       (TPA$_LAMBDA, ENDPRO)  
       );  
  
$STATE (ENDPRO,  
       ('', ' ', NEXT_PRO),  
       (')', ' ', OPTIONS)  
       );
```

```
!+  
! Subexpression to parse a UIC string.  
!-
```

# LIB\$TPARSE

```
$STATE (UIC,
        ('[')
        );

$STATE (,
        (TPA$_OCTAL,,, UIC_GROUP)
        );

$STATE (,
        ('', ')
        );

$STATE (,
        (TPA$_OCTAL,,, UIC_MEMBER)
        );

$STATE (,
        (']', TPA$_EXIT, CHECK_UIC)
        );

!+
! Subexpression to parse a general directory string
!-

$STATE (NAME,
        ('[')
        );

$STATE (NAMEO,
        (TPA$_STRING,, STORE_NAME)
        );

$STATE (,
        ('.', NAMEO),
        (']', TPA$_EXIT)
        );

PSECT OWN = $OWN$;
PSECT GLOBAL = $GLOBAL$;

GLOBAL ROUTINE CREATE_DIR (START_ADDR, CLI_CALLBACK) =
BEGIN
!+
! This program creates a directory. It gets the command
! line from the CLI and parses it with TPARSE.
!-

LOCAL
    STATUS,                ! Status from LIB$TPARSE
    OUT_LEN : WORD;        ! length of returned command line

EXTERNAL
    SS$_NORMAL;

EXTERNAL ROUTINE
    LIB$GET_FOREIGN : ADDRESSING_MODE (GENERAL),
    LIB$TPARSE      : ADDRESSING_MODE (GENERAL);

    COMMAND_DESC [DSC$W_LENGTH] = 256;
    COMMAND_DESC [DSC$B_DTYPE]  = DSC$K_DTYPE_T;
    COMMAND_DESC [DSC$B_CLASS]  = DSC$K_CLASS_S;
    COMMAND_DESC [DSC$A_POINTER] = COMMAND_BUFF;
```

# LIB\$TPARSE

```
STATUS = LIB$GET_FOREIGN (COMMAND_DESC,
                          %ASCID'COMMAND: ',
                          OUT_LEN
                          );

IF NOT .STATUS
  THEN
    SIGNAL (STATUS);

!+
! Copy the input string descriptor into the TPARSE control block
! and call TPARSE. Note that impure storage is assumed to be zero.
!-

TPARSE_BLOCK[TPA$L_STRINGCNT] = .OUT_LEN;
TPARSE_BLOCK[TPA$L_STRINGPTR] = .COMMAND_DESC[DSC$A_POINTER];

STATUS = LIB$TPARSE (TPARSE_BLOCK, .UFD_STATE, UFD_KEY);
IF NOT .STATUS
  THEN
    RETURN 0;
RETURN SS$_NORMAL
END;                                     ! End of routine CREATE_DIR

!+
! Parser action routines
!-

!+
! Shut off explicit blank processing after passing the command name.
!-

ROUTINE BLANKS_OFF =
  BEGIN
    TPARSE_ARGS;

    AP[TPA$V_BLANKS] = 0;
    1
  END;

!+
! Check the UIC for legal value range.
!-

ROUTINE CHECK_UIC =
  BEGIN
    TPARSE_ARGS;

    IF .UIC_GROUP<16,16> NEQ 0
      OR .UIC_MEMBER<16,16> NEQ 0
      THEN RETURN 0;

    FILE_OWNER<0,16> = .UIC_MEMBER;
    FILE_OWNER<16,16> = .UIC_GROUP;
    1
  END;

!+
! Store a directory name component.
!-

ROUTINE STORE_NAME =
  BEGIN
    TPARSE_ARGS;
```



# LIB\$TPARSE

```
IF .NAME_COUNT GEQU 8
OR .AP[TPA$L_TOKENCNT] GTRU 9
THEN RETURN 0;
NAME_COUNT = .NAME_COUNT + 1;
NAME_VECTOR [.NAME_COUNT, STRING_COUNT] = .AP[TPA$L_TOKENCNT];
NAME_VECTOR [.NAME_COUNT, STRING_ADDR] = .AP[TPA$L_TOKENPTR];
1
END;

!+
! Convert a UIC into its equivalent directory file name.
!-

ROUTINE MAKE_UIC =
BEGIN
TPARSE_ARGS;

IF .UIC_GROUP<8,8> NEQ 0
OR .UIC_MEMBER<8,8> NEQ 0
THEN RETURN 0;
DIRNAME1[0] = 0;
DIRNAME1[1] = UIC_STRING;
$FAOL (CTRSTR = UPLIT (6, UPLIT BYTE ('!OB!OB')),
      OUTBUF = DIRNAME1,
      PRMLST = UIC_GROUP
    );
1
END;
END
ELUDOM                                ! End of module CREATE_DIR
```

This BLISS program accepts and parses the command line of a  
CREATE/DIRECTORY command.

2

```
.TITLE          CREATE_DIR - Create Directory File
.IDENT          "X0000"

;+
;
; This is a sample program that accepts and parses the command line
; of the CREATE/DIRECTORY command. This program contains the VMS
; call to acquire the command line from the command interpreter
; and parse it with TPARSE, leaving the necessary information in
; its global data base. The command line has the following format:
;
;
;     CREATE/DIR DEVICE: [MARANTZ.ACCOUNT.OLD]
;                   /OWNER_UIC=[2437,25]
;                   /ENTRIES=100
;                   /PROTECTION=(SYSTEM:R, OWNER:RWED, GROUP:R, WORLD:R)
;
; The three qualifiers are optional. Alternatively, the command
; may take the form
;
;     CREATE/DIR DEVICE: [202,31]
;
; using any of the optional qualifiers.
;-
```

# LIB\$TPARSE

```
;+
;
; Global data, control blocks, etc.
;
;-
        .PSECT  IMPURE,WRT,NOEXE
;+
; Define control block offsets
;-
        $CLIDEF
        $TPADEF
;+
; Define parser flag bits for flags longword
;-
UIC_FLAG          = 1          ; /UIC seen
ENTRIES_FLAG      = 2          ; /ENTRIES seen
PROT_FLAG         = 4          ; /PROTECTION seen
;+
; LIB$GET_FOREIGN string descriptors to get the line to be parsed
;-
STRING_LEN = 256
STRING_DESC:
        .WORD STRING_LEN
        .BYTE DSC$K_DTYPE_T
        .BYTE DSC$K_CLASS_S
        .ADDRESS STRING_AREA
STRING_AREA:
        .BLKB STRING_LEN
PROMPT_DESC:
        .WORD PROMPT_LEN
        .BYTE DSC$K_DTYPE_T
        .BYTE DSC$K_CLASS_S
        .ADDRESS PROMPT
PROMPT:
        .ASCII /qualifiers: /
PROMPT_LEN = .-PROMPT
;+
; TPARSE argument block
;-
TPARSE_BLOCK:
        .LONG          TPA$K_COUNTO          ; Longword count
        .LONG          TPA$M_ABBREV!-        ; Allow abbreviation
        .LONG          TPA$M_BLANKS          ; Process spaces explicitly
        .BLKB          TPA$K_LENGTHO-8       ; Remainder set at run time
;+
; Parser global data
;-
```

# LIB\$TPARSE

```
RET_LEN:          .BLKW      1      ; LENGTH OF RETURNED COMMAND LINE
PARSER_FLAGS:     .BLKL      1      ; Keyword flags
DEVICE_STRING:    .BLKL      2      ; Device string descriptor
ENTRY_COUNT:      .BLKL      1      ; Space to preallocate
FILE_PROTECT:     .BLKL      1      ; Directory file protection
UIC_GROUP:        .BLKL      1      ; Temp for UIC group
UIC_MEMBER:       .BLKL      1      ; Temp for UIC member
UIC_STRING:       .BLKB      6      ; String to receive converted UIC
FILE_OWNER:       .BLKL      1      ; Actual file owner UIC
NAME_COUNT:       .BLKL      1      ; Number of directory names
DIRNAME1:         .BLKL      2      ; Name descriptor 1
DIRNAME2:         .BLKL      2      ; Name descriptor 2
DIRNAME3:         .BLKL      2      ; Name descriptor 3
DIRNAME4:         .BLKL      2      ; Name descriptor 4
DIRNAME5:         .BLKL      2      ; Name descriptor 5
DIRNAME6:         .BLKL      2      ; Name descriptor 6
DIRNAME7:         .BLKL      2      ; Name descriptor 7
DIRNAME8:         .BLKL      2      ; Name descriptor 8
```

.SBTTL Main Program

```
;+
; This program gets the CREATE/DIRECTORY command line from
; the command interpreter and parses it.
;-
.PSECT CODE,EXE,NOWRT
CREATE_DIR::
.WORD ^M<R2,R3,R4,R5> ; Save registers
;+
; Call the command interpreter to obtain the command line.
;-
.PUSHAW RET_LEN
.PUSHAQ PROMPT_DESC
.PUSHAQ STRING_DESC
CALLS #3,G^LIB$GET_FOREIGN ; Call to get command line
BLBC RO, SYNTAX_ERR
;+
; Copy the input string descriptor into the TPARSE control block
; -and call LIB$TPARSE. Note that impure storage is assumed to be zero.
;
MOVZWL RET_LEN, TPARSE_BLOCK+TPA$L_STRINGCNT
MOVAL STRING_AREA, TPARSE_BLOCK+TPA$L_STRINGPTR
PUSHAL UFD_KEY
PUSHAL UFD_STATE
PUSHAL TPARSE_BLOCK
CALLS #3,G^LIB$TPARSE
BLBC RO,SYNTAX_ERR
;+
; Parsing is complete.
;
; You can include here code to process the string just parsed, to call
; another program to process the command, or to return control to
; a calling program, if any.
;-
SYNTAX_ERR:
;+
; Code to handle parsing errors.
;-
```

RET

.SBTTL Parser State Table

# LIB\$TPARSE

```
;+
; Assign values for protection flags to be used when parsing protection
; string.
;-

SYSTEM_READ_FLAG = ^X0001
SYSTEM_WRITE_FLAG = ^X0002
SYSTEM_EXECUTE_FLAG = ^X0004
SYSTEM_DELETE_FLAG = ^X0008
GROUP_READ_FLAG = ^X0001
GROUP_WRITE_FLAG = ^X0002
GROUP_EXECUTE_FLAG = ^X0004
GROUP_DELETE_FLAG = ^X0008
OWNER_READ_FLAG = ^X0001
OWNER_WRITE_FLAG = ^X0002
OWNER_EXECUTE_FLAG = ^X0004
OWNER_DELETE_FLAG = ^X0008
WORLD_READ_FLAG = ^X0001
WORLD_WRITE_FLAG = ^X0002
WORLD_EXECUTE_FLAG = ^X0004
WORLD_DELETE_FLAG = ^X0008

$INIT_STATE      UFD_STATE,UFD_KEY

;+
; Read over the command name (to the first blank in the command).
;-
        $STATE      START
        $TRAN      TPA$_BLANK,,BLANKS_OFF
        $TRAN      TPA$_ANY,START

;+
; Read device name string and trailing colon.
;-
        $STATE
        $TRAN      TPA$_SYMBOL,,,,DEVICE_STRING

        $STATE
        $TRAN      ':'

;+
; Read directory string, which is either a UIC string or a general
; directory string.
;-
        $STATE
        $TRAN      !UIC,,MAKE_UIC
        $TRAN      !NAME

;+
; Scan for options until end of line is reached
;-
        $STATE      OPTIONS
        $TRAN      '/'
        $TRAN      TPA$_EOS,TPA$_EXIT

        $STATE
        $TRAN      'OWNER_UIC',PARSE_UIC,,UIC_FLAG,PARSER_FLAGS
        $TRAN      'ENTRIES',PARSE_ENTRIES,,ENTRIES_FLAG,PARSER_FLAGS
        $TRAN      'PROTECTION',PARSE_PROT,,PROT_FLAG,PARSER_FLAGS

;+
; Get file owner UIC.
;-
        $STATE      PARSE_UIC
        $TRAN      ':'
        $TRAN      '='
```

# LIB\$TPARSE

```

    $STATE
    $TRAN      !UIC,OPTIONS

;+
; Get number of directory entries.
;-

    $STATE      PARSE_ENTRIES
    $TRAN      ':'
    $TRAN      '='

    $STATE
    $TRAN      TPA$_DECIMAL,OPTIONS,,ENTRY_COUNT

;+
; Get directory file protection. Note that the bit masks generate the
; protection in complement form. It will be uncomplemented by the main
; program.
;-

    $STATE      PARSE_PROT
    $TRAN      ':'
    $TRAN      '='

    $STATE
    $TRAN      '('

    $STATE      NEXT_PRO
    $TRAN      'SYSTEM', SYPR
    $TRAN      'OWNER', OWPR
    $TRAN      'GROUP', GRPR
    $TRAN      'WORLD', WOPR

    $STATE      SYPR
    $TRAN      ':'
    $TRAN      '='

    $STATE      SYPRO
    $TRAN      'R',SYPRO,,SYSTEM_READ_FLAG,FILE_PROTECT
    $TRAN      'W',SYPRO,,SYSTEM_WRITE_FLAG,FILE_PROTECT
    $TRAN      'E',SYPRO,,SYSTEM_EXECUTE_FLAG,FILE_PROTECT
    $TRAN      'D',SYPRO,,SYSTEM_DELETE_FLAG,FILE_PROTECT
    $TRAN      TPA$_LAMBDA,ENDPRO

    $STATE      OWPR
    $TRAN      ':'
    $TRAN      '='

    $STATE      OWPRO
    $TRAN      'R',OWPRO,,OWNER_READ_FLAG,FILE_PROTECT
    $TRAN      'W',OWPRO,,OWNER_WRITE_FLAG,FILE_PROTECT
    $TRAN      'E',OWPRO,,OWNER_EXECUTE_FLAG,FILE_PROTECT
    $TRAN      'D',OWPRO,,OWNER_DELETE_FLAG,FILE_PROTECT
    $TRAN      TPA$_LAMBDA,ENDPRO

    $STATE      GRPR
    $TRAN      ':'
    $TRAN      '='

    $STATE      GRPRO
    $TRAN      'R',GRPRO,,GROUP_READ_FLAG,FILE_PROTECT
    $TRAN      'W',GRPRO,,GROUP_WRITE_FLAG,FILE_PROTECT
    $TRAN      'E',GRPRO,,GROUP_EXECUTE_FLAG,FILE_PROTECT
    $TRAN      'D',GRPRO,,GROUP_DELETE_FLAG,FILE_PROTECT
    $TRAN      TPA$_LAMBDA,ENDPRO

    $STATE      WOPR
    $TRAN      ':'
    $TRAN      '='

```

# LIB\$TPARSE

```

$STATE      WOPRO
$TRAN       'R',WOPRO,,WORLD_READ_FLAG,FILE_PROTECT
$TRAN       'W',WOPRO,,WORLD_WRITE_FLAG,FILE_PROTECT
$TRAN       'E',WOPRO,,WORLD_EXECUTE_FLAG,FILE_PROTECT
$TRAN       'D',WOPRO,,WORLD_DELETE_FLAG,FILE_PROTECT
$TRAN       TPA$_LAMBDA,ENDPRO

$STATE      ENDPRO
$TRAN       <','>,NEXT_PRO
$TRAN       ')' ,OPTIONS

;+
; Subexpression to parse a UIC string.
;-

$STATE      UIC
$TRAN       '['

$STATE
$TRAN       TPA$_OCTAL,,,UIC_GROUP

$STATE
$TRAN       <','>      ; The comma character must be
                       ;   surrounded by angle brackets
                       ;   because MACRO restricts the use
                       ;   of commas in arguments to macros.

$STATE
$TRAN       TPA$_OCTAL,,,UIC_MEMBER

$STATE
$TRAN       ']' ,TPA$_EXIT,CHECK_UIC

;+
; Subexpression to parse a general directory string.
;-

$STATE      NAME
$TRAN       '['

$STATE      NAMED
$TRAN       TPA$_STRING,,STORE_NAME

$STATE
$TRAN       ' ',NAMED
$TRAN       ']' ,TPA$_EXIT
$END_STATE

.SBTTL      Parser Action Routines
.PSECT      CODE,EXE,NOWRT

;+
; Shut off explicit blank processing after passing the command name.
;-

BLANKS_OFF:
.WORD      0          ; No registers saved (or used)
BCC        #TPA$_BLANKS,TPA$_L_OPTIONS(AP),10$
10$:      RET

;+
; Check the UIC for legal value range.
;-

```

# LIB\$TPARSE

```

CHECK_UIC:
    .WORD      0                ; No registers saved (or used)
    TSTW      UIC_GROUP+2      ; UIC components are 16 bits
    BNEQ      10$
    TSTW      UIC_MEMBER+2
    BNEQ      10$
    MOVW      UIC_GROUP,FILE_OWNER+2 ; Store actual UIC
    MOVW      UIC_MEMBER,FILE_OWNER ; after checking
    RET
10$:  CLRL      RO                ; Value out of range - fail
    RET                ; the transition

;+
; Store a directory name component.
;-

STORE_NAME:
    .WORD      0                ; No registers saved (or used)
    MOVL      NAME_COUNT,R1      ; Get count of names so far
    CMLP      R1,#8              ; Maximum of 8 permitted
    BGEQU     10$
    INCL      NAME_COUNT         ; Count this name
    MOVAQ     DIRNAME1[R1],R1    ; Address of next descriptor
    MOVQ      TPA$L_TOKENCNT(AP),(R1) ; Store the descriptor
    CMLP      (R1),#9           ; Check the length of the name
    BGRU      10$               ; Maximum is 9
    RET
10$:  CLRL      RO                ; Error in directory name
    RET

;+
; Convert a UIC into its equivalent directory file name.
;-

MAKE_UIC:
    .WORD      0                ; No registers saved (or used)
    TSTB      UIC_GROUP+1      ; Check UIC for byte values,
    BNEQ      10$              ; Since UIC type directories
    TSTB      UIC_MEMBER+1    ; Are restricted to this form
    BNEQ      10$
    MOVL      #6,DIRNAME1      ; Directory name is 6 bytes
    MOVAL     UIC_STRING,DIRNAME1+4 ; Point to string buffer
    $FAOL     CTRSTR=FAO_STRING,- ; Convert UIC to octal string
            OUTBUF=DIRNAME1,-
            PRMLST=UIC_GROUP

    RET
10$:  CLRL      RO                ; Range error - fail it
    RET

FAO_STRING: .LONG      STRING_END-STRING_START
STRING_START: .ASCII  '!OB!OB'
STRING_END:

    .END      CREATE_DIR

```

This MACRO program accepts and parses the command line of a CREATE/DIRECTORY command.





# LIB\$TRA\_ASC\_EBC

## ASCII to EBCDIC Translation Table

- The number on the left represents the low-order bits of the ASCII character in hexadecimal notation.
- The number across the top represents the high-order bits of the ASCII character in hexadecimal notation.
- The number in the body of the table represents the equivalent EBCDIC character in hexadecimal notation.

**Table LIB-23 LIB\$AB\_ASC\_EBC**

Row bits 0 – 3	Column bits 4 – 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	40	F0	7C	D7	79	97	3F	3F	3F	3F	3F	3F	3F	3F
1	01	11	4F	F1	C1	D8	81	98	3F	3F	3F	3F	3F	3F	3F	3F
2	02	12	7F	F2	C2	D9	82	99	3F	3F	3F	3F	3F	3F	3F	3F
3	03	13	7B	F3	C3	E2	83	A2	3F	3F	3F	3F	3F	3F	3F	3F
4	37	3C	5B	F4	C4	E3	84	A3	3F	3F	3F	3F	3F	3F	3F	3F
5	2D	3D	6C	F5	C5	E4	85	A4	3F	3F	3F	3F	3F	3F	3F	3F
6	2E	32	50	F6	C6	E5	86	A5	3F	3F	3F	3F	3F	3F	3F	3F
7	2F	26	7D	F7	C7	E6	87	A6	3F	3F	3F	3F	3F	3F	3F	3F
8	16	18	4D	F8	C8	E7	88	A7	3F	3F	3F	3F	3F	3F	3F	3F
9	05	19	5D	F9	C9	E8	89	A8	3F	3F	3F	3F	3F	3F	3F	3F
A	25	3F	5C	7A	D1	E9	91	A9	3F	3F	3F	3F	3F	3F	3F	3F
B	0B	27	4E	5E	D2	4A	92	C0	3F	3F	3F	3F	3F	3F	3F	3F
C	0C	1C	6B	4C	D3	E0	93	6A	3F	3F	3F	3F	3F	3F	3F	3F
D	0D	1D	60	7E	D4	5A	94	D0	3F	3F	3F	3F	3F	3F	3F	3F
E	0E	1E	4B	6E	D5	5F	95	A1	3F	3F	3F	3F	3F	3F	3F	3F
F	0F	1F	61	6F	D6	6D	96	07	3F	3F	3F	3F	3F	3F	3F	FF

ZK-4246-85

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Routine successfully completed.

LIB\$\_INVCHA

One or more occurrences of an untranslatable character have been detected during the translation.

LIB\$\_INVARG

If the destination string is a fixed-length string and its length is not the same as the source string length, no translation is attempted.

**EXAMPLE**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. TRANS.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 INPUT-STRING PIC X(4).
01 EBCDIC-STRING PIC X(4).
01 OUT-STRING PIC X(4).
01 FILL-CHAR PIC X VALUE "@".
01 SS-STATUS PIC S9(9) COMP.
88 SS-NORMAL VALUE 01.

01 EBCDIC-TABLE.
05 FILLER PIC X(16) VALUE "aaaaaaaaaaaaaaaa".
05 FILLER PIC X(16) VALUE "bbbbbbbbbbbbbbbb".
05 FILLER PIC X(16) VALUE "cccccccccccccccc".
05 FILLER PIC X(16) VALUE "dddddddddddddddd".
05 FILLER PIC X(16) VALUE " eeeeeeeeeee.<(+|"
05 FILLER PIC X(16) VALUE "&eeeeeeeeee!$*);@".
05 FILLER PIC X(16) VALUE "-/eeeeeeeeee,%_>?".
05 FILLER PIC X(16) VALUE "eeeeeeeeee:#@'=""".
05 FILLER PIC X(16) VALUE "@abcdefghi@@@@@".
05 FILLER PIC X(16) VALUE "@jklmnopqr@@@@@".
05 FILLER PIC X(16) VALUE "@stuvwxyz@@@@@".
05 FILLER PIC X(16) VALUE "cccccccccccccccc".
05 FILLER PIC X(16) VALUE "@ABCDEFGHII@@@@@".
05 FILLER PIC X(16) VALUE "!JKLMNOPQR@@@@@".
05 FILLER PIC X(16) VALUE "@@STUVWXYZ@@@@@".
05 FILLER PIC X(16) VALUE "0123456789@@@@@".

ROUTINE DIVISION.

001-MAIN.
DISPLAY " ".
DISPLAY "ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC: "
WITH NO ADVANCING
ACCEPT INPUT-STRING
AT END STOP RUN.
IF INPUT-STRING = "EXIT" OR "exit" OR " "
STOP RUN.

CALL "LIB$TRA_ASC_EBC"
USING BY DESCRIPTOR INPUT-STRING, EBCDIC-STRING
GIVING SS-STATUS.
IF SS-NORMAL
CALL "LIB$MOVTC"
USING BY DESCRIPTOR EBCDIC-STRING,
FILL-CHAR,
EBCDIC-TABLE,
OUT-STRING,
GIVING SS-STATUS
IF SS-NORMAL
DISPLAY "ASCII ENTERED WAS: " INPUT-STRING
DISPLAY "EBCDIC TRANSLATED IS: " OUT-STRING
ELSE
DISPLAY "*** LIB$MOVTC TRANSLATION UNSUCCESSFUL ***"
ELSE
DISPLAY "*** LIB$TRA_ASC_EBC TRANSLATION UNSUCCESSFUL ***".
GO TO 001-MAIN.

```

# LIB\$TRA\_ASC\_EBC

This COBOL program uses LIB\$TRA\_ASC\_EBC to translate an ASCII string to EBCDIC. If successful, it then uses LIB\$MOVTC to translate the EBCDIC string back to ASCII.

To exit from this program, you must type CTRL/Z. The output generated by this COBOL program is as follows:

```
$ RUN TRANS
```

```
ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC:  abdc  
ASCII ENTERED WAS:  abdc  
EBCDIC TRANSLATED IS:  abdc
```

```
ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC:  ~=b&  
ASCII ENTERED WAS:  ~=b&  
EBCDIC TRANSLATED IS:  @=b&
```

```
ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC:  8~%$  
ASCII ENTERED WAS:  8~%$  
EBCDIC TRANSLATED IS:  8@%$
```

```
ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC:  
/x\}  
ASCII ENTERED WAS:  /x\}  
EBCDIC TRANSLATED IS:  /x@!
```

```
ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC:  CTRL/Z
```



# LIB\$TRA\_EBC\_ASC

**Table LIB-24 LIB\$AB\_EBC\_ASC**

Row bits 0 - 3	Column bits 4 - 7															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	10	5C	5C	20	26	2D	5C	5C	5C	5C	7B	7D	5C	30	
1	01	11	5C	5C	5C	5C	2F	5C	61	6A	7E	5C	41	4A	5C	31
2	02	12	5C	16	5C	5C	5C	5C	62	6B	73	5C	42	4B	53	32
3	03	13	5C	5C	5C	5C	5C	5C	63	6C	74	5C	43	4C	54	33
4	5C	5C	5C	5C	5C	5C	5C	5C	64	6D	75	5C	44	4D	55	34
5	09	5C	0A	5C	5C	5C	5C	5C	65	6E	76	5C	45	4E	56	35
6	5C	08	17	5C	5C	5C	5C	5C	66	6F	77	5C	46	4F	57	36
7	7F	5C	1B	04	5C	5C	5C	5C	67	70	78	5C	47	50	58	37
8	5C	18	5C	5C	5C	5C	5C	5C	68	71	79	5C	48	51	59	38
9	5C	19	5C	5C	5C	5C	5C	60	69	72	7A	5C	49	52	5A	39
A	5C	5C	5C	5C	5B	5D	7C	3A	5C	5C	5C	5C	5C	5C	5C	5C
B	0B	5C	5C	5C	2E	24	2C	23	5C	5C	5C	5C	5C	5C	5C	5C
C	0C	1C	5C	14	3C	2A	25	40	5C	5C	5C	5C	5C	5C	5C	5C
D	0D	1D	05	15	28	29	5F	27	5C	5C	5C	5C	5C	5C	5C	5C
E	0E	1E	06	5C	2B	3B	3E	3D	5C	5C	5C	5C	5C	5C	5C	5C
F	0F	1F	07	1A	21	5E	3F	22	5C	5C	5C	5C	5C	5C	5C	FF

ZK-4249-85

## EBCDIC to ASCII Translation Table

- The number on the left represents the low-order bits of the EBCDIC character in hexadecimal notation.
- The number across the top represents the high-order bits of the EBCDIC character in hexadecimal notation.
- The number in the body of the table represents the equivalent ASCII character in hexadecimal notation.

## CONDITION VALUES RETURNED

SS\$\_NORMAL

Routine successfully completed.

LIB\$\_INVCHA

One or more occurrences of an untranslatable character have been detected during the translation.

LIB\$\_INVARG

If the destination string is a fixed-length string and its length is not the same as the source string length, no translation is attempted.



# LIB\$TRAVERSE\_TREE

---

## DESCRIPTION

LIB\$TRAVERSE\_TREE calls a user-supplied action routine for each node to traverse a balanced binary tree.

### Call Format for an Action Routine

The format of the call is as follows:

**user-action-procedure** treehead ,user-data-address

LIB\$TRAVERSE\_TREE passes the **treehead** and **user-data-address** arguments to your action routine by reference.

This action routine is defined by you to fit your own purposes. A common use of an action routine here is to print the contents of each node during the tree traversal.

This is one example of a user-supplied action routine.

```
1 %TITLE 'LIB$ Tree Example in BASIC V2'
  %SBTTL 'Function to display a node'

  FUNCTION LONG PRINT_NODE (NODE_TYPE NODE, LONG DUMMY)

    !+
    ! Print the string contained in the current node
    !-

    OPTION TYPE = EXPLICIT

    RECORD NODE_TYPE
      BYTE      HEADER (9)          ! Header
      BYTE      LENGTH             ! Length
      STRING    TEXT = 80          ! String
    END RECORD NODE_TYPE

    PRINT SEG$ (NODE::TEXT, 1%, NODE::LENGTH)

    PRINT_NODE = 1%
  END FUNCTION
```

---

## CONDITION VALUES RETURNED

LIB\$_NORMAL	Success. Traversal complete.
	Any condition value returned by your action routine.

---

## EXAMPLE

The BASIC example provided in the description of LIB\$INSERT\_TREE also demonstrates the use of LIB\$TRAVERSE\_TREE. Please refer to that example for assistance in using this routine.





# LIB\$TRIM\_FILESPEC

## *resultant-length*

VMS usage: **word\_unsigned**  
type: **word (unsigned)**  
access: **write only**  
mechanism: **by reference**

Length of the trimmed file specification, not including any blank padding or truncated characters. The **resultant-length** argument is the address of an unsigned word that contains this length. This is an optional argument.

---

## DESCRIPTION

This routine trims file specifications in a consistent, predictable manner to fit in a fixed-length field using the same algorithm that DIGITAL software uses.

LIB\$TRIM\_FILESPEC allows compilers and other utilities which need to display file specifications in fixed-length fields, such as listing headers, to display file specifications in a consistent fashion.

If necessary to make the file specification fit into the specified field width, LIB\$TRIM\_FILESPEC removes portions of the file specification in this order.

- 1 Node (including access control)
- 2 Device
- 3 Directory
- 4 Version
- 5 Type

If, after removing all these fields, the file name is still longer than the field width, the file name is truncated and the alternate success status LIB\$\_STRTRU is returned.

LIB\$TRIM\_FILESPEC supports any string class for the **old-filespec** and **new-filespec** string arguments.

---

## CONDITION VALUES RETURNED

SS\$_NORMAL	Routine successfully completed.
LIB\$_STRTRU	Success, but the output string was truncated. Significant characters of the trimmed file specification were truncated.
LIB\$_INVARG	Invalid argument. <b>Old-filespec</b> contained more than 255 characters.
LIB\$_INVSTRDES	Invalid string descriptor.
LIB\$_WRONUMARG	Wrong number of arguments.

Any condition values returned by LIB\$COPY\_R\_DX.

Any condition values returned by the \$FILESCAN system service.

**EXAMPLE**

```

PROGRAM TRIM_FILESPEC(INPUT,OUTPUT);

{+}
{ This Pascal example program demonstrates the
{ use of LIB$TRIM_FILESPEC.
{-}

    TYPE
        WORD = [WORD] 0..65535;

    VAR
        INPUT_FILESPEC : VARYING [255] OF CHAR;
        OUTPUT_FILESPEC : VARYING [32] OF CHAR;
        RETURNED_STATUS : INTEGER;

    [EXTERNAL] FUNCTION LIB$TRIM_FILESPEC(
        IN_FILE      : VARYING [LEN1] OF CHAR;
        VAR OUT_FILE : VARYING [LEN2] OF CHAR;
        WIDTH        : WORD := %IMMED 0;
        OUT_LEN      : [REFERENCE] WORD := %IMMED 0
    ) : INTEGER; EXTERNAL;

    [EXTERNAL] FUNCTION LIB$STOP(
        CONDITION_STATUS : [IMMEDIATE,UNSAFE] UNSIGNED;
        FAO_ARGS         : [IMMEDIATE,UNSAFE,LIST] UNSIGNED
    ) : INTEGER; EXTERNAL;

BEGIN

{+}
{ Start with a large INPUT_FILESPEC.
{-}

    INPUT_FILESPEC := 'DISK$NAME:[DIRECTORY1.DIRECTORY2]FILENAME.EXTENSTION;1';

{+}
{ Use LIB$TRIM_FILESPEC to shorten it to fit a smaller variable.
{-}

    RETURNED_STATUS := LIB$TRIM_FILESPEC(
        INPUT_FILESPEC,
        OUTPUT_FILESPEC,
        SIZE(OUTPUT_FILESPEC.BODY));

    IF NOT ODD(RETURNED_STATUS)
    THEN
        LIB$STOP(RETURNED_STATUS);

{+}
{ Print out the original file name along with the
{ shortened file name.
{-}

    WRITELN('Original file specification ',INPUT_FILESPEC);
    WRITELN('Shortened file specification ',OUTPUT_FILESPEC);

END.

```

This Pascal example program demonstrates the use of LIB\$TRIM\_FILESPEC. The output generated by this program is as follows:

```

Original file specification  DISK$NAME:[DIRECTORY1.DIRECTORY2]FILENAME.EXTENSTION;1
Shortened file specification  FILENAME.EXTENSTION;1

```

# LIB\$VERIFY\_VM\_ZONE

---

## LIB\$VERIFY\_VM\_ZONE Verify a Zone

The Verify a Zone routine performs verification of a zone.

---

**FORMAT**            **LIB\$VERIFY\_VM\_ZONE** *zone-id*

---

**RETURNS**            VMS usage: **cond\_value**  
                          type:       **longword (unsigned)**  
                          access:     **write only**  
                          mechanism: **by value**

---

**ARGUMENTS**        **zone-id**  
                          VMS usage: **identifier**  
                          type:       **longword (unsigned)**  
                          access:     **read only**  
                          mechanism: **by reference**

Zone identifier of the zone to be verified. The **zone-id** argument is the address of an unsigned longword that contains this zone identifier. A value of zero indicates the default zone.

---

**DESCRIPTION**        LIB\$VERIFY\_VM\_ZONE verifies a zone. LIB\$VERIFY\_VM\_ZONE performs verification of the zone header and scans all of the queues and lists maintained in the zone header; this includes the lookaside lists and the free lists. If the zone was created with LIB\$\_VM\_FREE\_FILL0 or LIB\$\_VM\_FREE\_FILL1, LIB\$VERIFY\_VM\_ZONE also checks the contents of the free blocks.

As soon as an error is encountered, processing stops. If LIB\$\_BADZONE is returned, use the routine LIB\$SHOW\_VM\_ZONE to dump the zone information.

You must have exclusive access to the zone while the verification is proceeding. Results are unpredictable if another thread of control modifies the zone while this routine is analyzing control data or scanning control blocks.

---

<b>CONDITION VALUES RETURNED</b>	SS\$_NORMAL	Normal successful completion.
	LIB\$_BADZONE	Invalid zone.
	LIB\$_INVARG	Invalid or null argument.
	LIB\$_WRONUMARG	Wrong number of arguments.

---

## LIB\$WAIT Wait a Specified Period of Time

The Wait a Specified Period of Time routine places the current process into hibernation for the number of seconds specified in its argument.

---

**FORMAT**            **LIB\$WAIT**    *seconds*

---

**RETURNS**            VMS usage: **cond\_value**  
                           type:            **longword (unsigned)**  
                           access:        **write only**  
                           mechanism:    **by value**

---

**ARGUMENT**            ***seconds***  
                           VMS usage: **floating\_point**  
                           type:         **F\_floating**  
                           access:       **read only**  
                           mechanism:   **by reference**

The number of seconds to wait. The **seconds** argument contains the address of an F-floating number that is this number.

The value is rounded to the nearest hundredth-second before use. Seconds must be between 0.0 and 100,000.0.

---

**DESCRIPTION**        LIB\$WAIT rounds the value specified by **seconds** to the nearest hundredth-second, uses the \$SCHDWK system service to schedule a wakeup for that interval, and then issues the \$HIBER system service to hibernate until the wakeup occurs.

Due to other system activity, the length of time that the process actually waits may be somewhat longer than what was specified by **seconds**.

The process hibernates in the caller's access mode; therefore, asynchronous system traps (ASTs) may be delivered while the process is hibernating. However, if the process hibernates at AST level, further ASTs can not be delivered. See the *VMS System Services Reference Manual* for more information.

---

**CONDITION  
VALUES  
RETURNED**

SS\$_NORMAL	Routine successfully completed.
LIB\$_INVARG	Invalid argument. The value of <b>seconds</b> was less than zero or was greater than 100,000.0
LIB\$_WRONUMARG	Wrong number of arguments. An incorrect number of arguments was passed to LIB\$WAIT.

Any condition values returned by the \$SCHDWK system service.

# LIB\$WAIT

---

## EXAMPLE

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. SAMPLE.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01    DELAY COMP-1.  
ROUTINE DIVISION.  
START-SAMPLE.  
    MOVE 3.5 TO DELAY.  
    CALL "LIB$WAIT"  
    USING BY REFERENCE DELAY.  
    STOP RUN.
```

This COBOL program demonstrates the use of LIB\$WAIT. When run, it waits for 3.5 seconds and then exits.

---

# Index

---

## A

---

Addition  
quadword times • LIB-5  
two's complement • LIB-7

Algorithm  
for memory allocation • 5-7

Alignment attribute • 5-11

Area extension size • 5-9

\$ASCTIM  
RTL jacket routine • LIB-401

AST (asynchronous system trap) • 2-22

## B

---

Bit field  
replace field • LIB-253  
return sign extended to longword • LIB-142

Block size • 5-10

Boundary tag • 5-8

## C

---

CALLG (Call Routine with General Argument List)  
instruction  
RTL routine to access • LIB-23

Chaining • 2-5

Channel • 2-23

Character string routine • 2-14  
LIB\$CHAR • LIB-25

Character string translation routine • 2-14

CLI (command language interpreter) • 2-2

CLI access routine • 2-2

CLI symbol • LIB-343  
deleting • LIB-116  
getting value of • LIB-219  
RTL routines • LIB-116, LIB-219

Command language interpreter  
See CLI

Condition handler • 4-12  
See also Signal argument vector  
catch-all • 4-14

Condition handler (cont'd.)  
continuing execution of • 4-21  
default • 4-13  
establishment of • 4-20, LIB-140  
interaction between default and user-supplied  
handlers • 4-15  
last-chance • 4-14  
resignaling • 4-21  
software supplied • 4-13  
traceback • 4-13  
unwinding • 4-22  
user-supplied • 4-13  
writing of • 4-20

Condition handling • 4-2  
See also Condition handler  
See also Condition Handling Facility  
See also Condition value  
See also Exception  
See also Exception condition  
See also Message Utility (MESSAGE)  
continuing • 4-14  
displaying messages • 4-16  
logging error messages • 4-4  
logging error messages to a file • 4-27  
resignaling • 4-14  
stack traceback • 4-3  
stack unwind • 4-4, 4-14  
user-defined messages • 4-4

Condition Handling Facility • 4-19  
defined • 4-1  
function of • 4-2

Condition value • 4-5 to 4-7, 4-24, LIB-272  
severity • 4-6

Conversion  
numeric text to binary • LIB-76

\$CRFCTLTABLE macro • 8-1, 8-2

\$CRFFIELDEND macro • 8-1, 8-4

\$CRFFIELD macro • 8-1, 8-3

Cross-Reference Routines • 8-1

Cyclic redundancy check table • LIB-33

---

## D

---

Date/Time routine  
LIB\$DATE\_TIME • LIB-80

## Index

Date/Time routine (cont'd.)  
LIB\$DAY • LIB-82  
LIB\$DAY\_OF\_WEEK • LIB-84  
Debugging programs that use VM zones • 6-1  
Decimal overflow detection • LIB-104  
Decimal text  
    converting to binary • LIB-76  
Directory  
    creation of • LIB-36  
Division  
    extended precision • LIB-126  
Dynamic memory allocation • 5-1

---

## E

---

EDIV (Extended Divide) instruction  
    RTL routine to access • LIB-126  
EMODD instruction  
    RTL routine to access • LIB-128  
EMODF instruction  
    RTL routine to access • LIB-130  
EMODG instruction  
    RTL routine to access • LIB-132  
EMODH instruction  
    RTL routine to access • LIB-134  
EMUL (Extended Multiply) instruction  
    RTL routine to access • LIB-136  
Error  
    signaling of • 4-3  
Event flag  
    allocation of • 2-17  
    RTL routine to free • LIB-174  
Exception  
    definition • 4-2  
    floating-point underflow • 4-31  
    how handled by Run-Time Library • 4-30  
Exception condition • 4-2, 4-4  
    returning condition value • 4-4  
    signaling of • 4-3, 4-5, 4-7, 4-16, 4-18,  
    4-23, 4-31

---

## F

---

\$FAO • 4-13, 4-16, 4-27  
    RTL jacket routine for • LIB-404  
Fault  
    fix floating reserved operand • LIB-165

FFx instruction  
    RTL routine to access • LIB-147  
Floating-point underflow • 4-31  
Foreign command • 2-3  
Foreign command name  
    use of dollar sign • 2-4

---

## G

---

\$GETMSG • 4-16

---

## H

---

Hexadecimal text  
    converting to binary • LIB-76  
Hibernation  
    LIB\$WAIT • LIB-465

---

## I

---

Integer and floating-point routine • 2-12  
Integer Overflow • LIB-255

---

## J

---

Jacket routine • 2-1

---

## K

---

Keyword  
    in keyword table • LIB-261

---

## L

---

LIB\$ADAWI • LIB-3  
LIB\$ADDX • LIB-7  
LIB\$ADD\_TIMES • LIB-5  
LIB\$ANALYZE\_SDESC • LIB-10  
LIB\$ASN\_WTH\_MBX • 2-23, LIB-12  
LIB\$AST\_IN\_PROG • 2-22, LIB-15

LIB\$ATTACH•2-9, LIB-17  
 LIB\$BBCCI•LIB-19  
 LIB\$BBSSI•LIB-21  
 LIB\$CALLG•2-16, LIB-23  
 LIB\$CHAR•LIB-25  
 LIB\$CONVERT\_DATE\_STRING•LIB-27  
 LIB\$CRC•2-16, LIB-31  
 LIB\$CRC\_TABLE•2-16, LIB-33  
 LIB\$CREATE\_DIR•2-24, LIB-36  
 LIB\$CREATE\_USER\_VM\_ZONE•5-12, 5-17,  
 LIB-40  
 LIB\$CREATE\_VM\_ZONE•5-6, 5-16, LIB-44  
 LIB\$CRF\_INS\_KEY•8-1, LIB-50  
 LIB\$CRF\_INS\_REF•8-1, LIB-52  
 LIB\$CRF\_OUTPUT•8-1, LIB-55  
 LIB\$CURRENCY•LIB-59  
 LIB\$CVTF\_FROM\_INTERNAL\_TIME•LIB-70  
 LIB\$CVTF\_TO\_INTERNAL\_TIME•LIB-74  
 LIB\$CVT\_DTB•LIB-76  
 LIB\$CVT\_DX\_DX•LIB-61  
 LIB\$CVT\_FROM\_INTERNAL\_TIME•LIB-67  
 LIB\$CVT\_HTB•LIB-76  
 LIB\$CVT\_OTB•LIB-76  
 LIB\$CVT\_TO\_INTERNAL\_TIME•LIB-72  
 LIB\$CVT\_VECTIM•LIB-78  
 LIB\$DATE\_TIME•LIB-80  
 LIB\$DAY•LIB-82  
 LIB\$DAY\_OF\_WEEK•LIB-84  
 LIB\$DECODE\_FAULT•4-30, LIB-86  
 LIB\$DEC\_OVER•4-32, LIB-104  
 LIB\$DELETE\_FILE•LIB-106  
 LIB\$DELETE\_LOGICAL•2-8, LIB-114  
 LIB\$DELETE\_SYMBOL•2-8, LIB-116  
 LIB\$DELETE\_VM\_ZONE•5-6, LIB-118  
 LIB\$DIGIT\_SEP•LIB-120  
 LIB\$DISABLE\_CTRL•2-9, LIB-122  
 LIB\$DO\_COMMAND•2-6, LIB-124  
 LIB\$EDIV•LIB-126  
 LIB\$EMODD•LIB-128  
 LIB\$EMODF•LIB-130  
 LIB\$EMODG•LIB-132  
 LIB\$EMODH•LIB-134  
 LIB\$EMUL•LIB-136  
 LIB\$ENABLE\_CTRL•2-9, LIB-138  
 LIB\$ESTABLISH•4-3, 4-13, 4-20, LIB-140  
 LIB\$EXTV•LIB-142  
 LIB\$EXTZV•LIB-145  
 LIB\$FFC•LIB-147  
 LIB\$FFS•LIB-147  
 LIB\$FID\_TO\_NAME•LIB-149  
 LIB\$FILE\_SCAN•LIB-151  
 LIB\$FILE\_SCAN\_END•LIB-153  
 LIB\$FIND\_FILE•LIB-155  
 LIB\$FIND\_FILE\_END•LIB-159  
 LIB\$FIND\_IMAGE\_SYMBOL•LIB-160  
 LIB\$FIND\_VM\_ZONE•5-6, LIB-163  
 LIB\$FIXUP\_FLT•4-30, LIB-165  
 LIB\$FLT\_UNDER•4-32, LIB-167  
 LIB\$FORMAT\_DATE\_TIME•LIB-169  
 LIB\$FREE\_DATE\_TIME\_CONTEXT•LIB-172  
 LIB\$FREE\_EF•LIB-174  
 LIB\$FREE\_LUN•LIB-175  
 LIB\$FREE\_TIMER•LIB-176  
 LIB\$FREE\_VM•5-3, LIB-177  
 LIB\$FREE\_VM\_PAGE•5-3, LIB-179  
 LIB\$GETDVI•LIB-181  
 LIB\$GETJPI•LIB-186  
 LIB\$GETQUI•LIB-191  
 LIB\$GETSYI•LIB-196  
 LIB\$GET\_COMMAND•LIB-199  
 LIB\$GET\_COMMON•2-5, 2-35, LIB-202  
 LIB\$GET\_DATE\_FORMAT•LIB-204  
 LIB\$GET\_EF•LIB-206  
 LIB\$GET\_FOREIGN•2-3, LIB-208  
 LIB\$GET\_INPUT•LIB-212  
 LIB\$GET\_LUN•LIB-215  
 LIB\$GET\_MAXIMUM\_DATE\_LENGTH•LIB-216  
 LIB\$GET\_SYMBOL•2-8, LIB-219  
 LIB\$GET\_USERS\_LANGUAGE•LIB-222  
 LIB\$GET\_VM•5-3, LIB-223  
 LIB\$GET\_VM\_PAGE•5-3, LIB-225  
 LIB\$ICHAR•LIB-227  
 LIB\$INDEX•LIB-229  
 LIB\$INITIALIZE•7-1  
 LIB\$INIT\_DATE\_TIME\_CONTEXT•LIB-231  
 LIB\$INIT\_TIMER•LIB-235  
 LIB\$INSERT\_TREE•2-31, LIB-237  
 LIB\$INSQHI•LIB-248  
 LIB\$INSQTI•LIB-251  
 LIB\$INSV•LIB-253  
 LIB\$INT\_OVER•4-32, LIB-255  
 LIB\$LEN•LIB-257  
 LIB\$LOCC•LIB-258  
 LIB\$LOOKUP\_KEY•LIB-261  
 LIB\$LOOKUP\_TREE•2-31, LIB-265  
 LIB\$LPL\_LINES•LIB-267  
 LIB\$MATCHC•LIB-270  
 LIB\$MATCH\_COND•4-10, 4-30, LIB-272  
 LIB\$MOV3•LIB-275  
 LIB\$MOV5•LIB-276  
 LIB\$MOVTC•LIB-278  
 LIB\$MOVUC•LIB-295



## Index

LIB\$MULTF\_DELTA\_TIME • LIB-298  
LIB\$MULT\_DELTA\_TIME • LIB-297  
LIB\$PAUSE • LIB-299  
LIB\$POLYD • LIB-300  
LIB\$POLYF • LIB-302  
LIB\$POLYG • LIB-305  
LIB\$POLYH • LIB-307  
LIB\$PUT\_COMMON • 2-5, 2-35, LIB-309  
LIB\$PUT\_OUTPUT • LIB-311  
LIB\$RADIX\_POINT • LIB-313  
LIB\$REMQHI • LIB-315  
LIB\$REMQTI • LIB-317  
LIB\$RENAME\_FILE • LIB-319  
LIB\$RESERVE\_EF • LIB-327  
LIB\$RESET\_VM\_ZONE • 5-13, 5-14, LIB-329  
LIB\$REVERT • 4-3, 4-20, LIB-331  
LIB\$RUN\_PROGRAM • 2-5, LIB-332  
LIB\$SCANC • LIB-334  
LIB\$SCOPY\_DXDX • LIB-336  
LIB\$SCOPY\_R\_DX • LIB-338  
LIB\$SET\_LOGICAL • 2-8, LIB-340  
LIB\$SET\_SYMBOL • 2-8, LIB-343  
LIB\$SFREE1\_DD • LIB-347  
LIB\$SFREEN\_DD • LIB-348  
LIB\$SGET1\_DD • LIB-350  
LIB\$SHOW\_TIMER • LIB-352  
LIB\$SHOW\_VM • LIB-356  
LIB\$SHOW\_VM\_ZONE • 5-6, LIB-359  
LIB\$SIGNAL • 4-2, 4-3, 4-7, 4-10, 4-11,  
4-12, 4-14, 4-16, 4-22, 4-23 to 4-26,  
4-31, LIB-365  
LIB\$SIG\_TO\_RET • 4-29, LIB-369  
LIB\$SIG\_TO\_STOP • 4-29, LIB-372  
LIB\$SIM\_TRAP • 4-21, 4-29, LIB-374  
LIB\$SKPC • LIB-376  
LIB\$SPANC • LIB-378  
LIB\$SPAWN • 2-9, LIB-382  
LIB\$STAT\_TIMER • LIB-388  
LIB\$STAT\_VM • LIB-392  
LIB\$STOP • 4-2, 4-3, 4-4, 4-7, 4-10, 4-12,  
4-14, 4-16, 4-21, 4-22, 4-23 to 4-26,  
LIB-394  
LIB\$SUBX • LIB-399  
LIB\$SUB\_TIMES • LIB-397  
LIB\$SYS\_ASCTIM • LIB-401  
LIB\$SYS\_FAO • LIB-404  
LIB\$SYS\_FAOL • LIB-406  
LIB\$SYS\_GETMSG • LIB-408  
LIB\$TPARSE • LIB-411  
LIB\$TRAVERSE\_TREE • 2-31, LIB-459  
LIB\$TRA\_ASC\_EBC • LIB-453

LIB\$TRA\_EBC\_ASC • LIB-457  
LIB\$TRIM\_FILESPEC • LIB-461  
LIB\$VERIFY\_VM\_ZONE • 5-6, LIB-464  
LIB\$WAIT • LIB-465  
Logical name • LIB-340  
    RTL routines • LIB-114  
Logical unit number  
    allocating • 2-17  
    RTL routine to free • LIB-175

---

## M

Mailbox • 2-23, LIB-12  
MATCHC (Match Characters) instruction  
    RTL routine to access • LIB-270  
Mechanism argument vector • 4-7, 4-11, 4-20  
Memory  
    allocating and freeing blocks of • 5-4  
    allocating and freeing pages of • 5-4  
    allocation algorithms • 5-7  
Memory fragmentation • 5-5  
Memory management system services • 5-3  
Message Utility (MESSAGE) • 4-26 to 4-28  
MOVC3 (Move Character 3 Operand) instruction  
    RTL routine to access • LIB-275  
MOVC5 (Move Character 5 Operand) instruction  
    RTL routine to access • LIB-276  
Multiplication • LIB-128, LIB-130, LIB-132,  
LIB-134  
    extended precision • LIB-136

---

## O

Octal text  
    converting to binary • LIB-76  
Output formatting control routine • 2-20

---

## P

Performance measurement routine • 2-18  
Polynomial  
    evaluating • LIB-300, LIB-302, LIB-305,  
LIB-307  
\$PUTMSG • 4-4, 4-13, 4-16, 4-27

---

## Q

---

- Queue • 2-12, LIB-251
    - self-relative • 2-13
  - Queue access routine • 2-13
  - Queues
    - entry insertion • LIB-248
- 

## R

---

- Reserved operand
    - fix floating-point fault • LIB-165
  - Routine
    - processwide resource allocation • 2-16, 2-17
    - variable-length bit field • 2-10
  - Run-Time Library
    - condition handling • 4-1
    - queue access • 2-12
  - Run-Time Library routine
    - integer and floating-point • 2-12
    - interaction with operating system • 2-1
    - jacket routine • 2-1
    - library • 1-1
    - output formatting control • 2-20
    - performance measurement • 2-18
    - system service access • 2-1
    - to access command language interpreter • 2-2
    - to access VAX instruction set • 2-9
    - to access VMS system components • 2-1
    - to manipulate character string • 2-14
    - variable-length bit field instruction • 2-10
- 

## S

---

- SCANC (SCAN Characters) instruction
  - RTL routine to access • LIB-334
- Shareable image
  - activating • LIB-160
- Signal argument vector • 4-7, 4-9, 4-20
- Sign-Extended longword field • LIB-142
- String
  - copying by descriptor • LIB-336
  - copying by reference • LIB-338
  - skipping characters in • LIB-379
- String descriptor • LIB-10

- Subprocess
    - connecting to using LIB\$ATTACH • 2-9
    - creation of using LIB\$SPAWN • 2-9
  - Subtraction
    - quadword times • LIB-397
    - two's complement • LIB-400
  - System service access • 2-1, 2-2
- 

## U

---

- \$UNWIND • 4-14, 4-21, 4-22 to 4-23, 4-29
- 

## V

---

- Variable-length bit field routine • 2-11
  - VAX instruction set
    - accessing through Run-Time Library • 2-9
- 

## Z

---

- Zone • 5-6
  - allocation algorithm • 5-15
  - attribute • 5-8
  - creating • 5-6
  - deleting • 5-6
  - identifier • 5-12
  - resetting • 5-14
  - the default zone • 5-12
  - user-created • 5-6



# Reader's Comments

VMS RTL Library  
(LIB\$) Manual  
AA-LA76A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I would like to see more/less \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

What I like best about this manual is \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

What I like least about this manual is \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

I am using **Version** \_\_\_\_\_ of the software this manual describes.

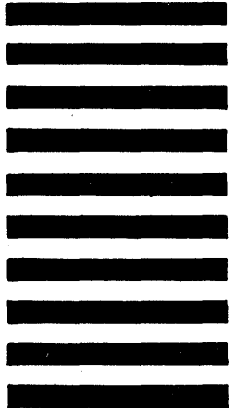
Name/Title \_\_\_\_\_ Dept. \_\_\_\_\_  
Company \_\_\_\_\_ Date \_\_\_\_\_  
Mailing Address \_\_\_\_\_  
\_\_\_\_\_ Phone \_\_\_\_\_

--- Do Not Tear - Fold Here and Tape ---

**digital**<sup>TM</sup>



No Postage  
Necessary  
if Mailed  
in the  
United States



**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION  
Corporate User Publications—Spit Brook  
ZK01-3/J35 110 SPIT BROOK ROAD  
NASHUA, NH 03062-9987



--- Do Not Tear - Fold Here ---